

Documentation Writing Guidelines

Best Practices

- Check the spelling and grammar, even if you have to copy and paste from an external source.
- Use simple sentences. Easy-to-read sentences mean the reader can quickly use the guidance you share.
- Try to express your thoughts in a concise and clean way.
- Don't abuse `code` format when writing in plain English.
- Follow Google developer documentation [style guide](#).
- Check the meaning of words in Microsoft's [A-Z word list and term collections](#) (use the search input!).
- RFC keywords should be used in technical documents (uppercase) and we recommend to use them in user documentation (lowercase). The RFC keywords are: "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL". They are to be interpreted as described in [RFC 2119](#).

Links

NOTE: Strongly consider the existing links - both within this directory and to the website docs - when moving or deleting files.

Relative links should be used nearly everywhere, due to versioning. Note that in case of page reshuffling, you must update all links references. When deleting a link, redirects must be created in

`docsaurus.config.js` to preserve the user flow.

Code Snippets

Code snippets can be included in the documentation using normal Markdown code blocks. For example:

```
```go
func() {}
```

It is also possible to include code snippets from GitHub files by referencing the files directly (and the line numbers if needed). For example:

```
```go reference
https://github.com/cosmos/cosmos-sdk/blob/v0.46.0/server/types/app.go#L57-L59
```
```

## Technical Writing Course

Google provides a free [course](#) for technical writing.

## Updating the docs

If you want to open a PR in Cosmos SDK to update the documentation, please follow the guidelines in [CONTRIBUTING.md](#) and the [Documentation Writing Guidelines](#).

## Stack

The documentation for Cosmos SDK is hosted at <https://docs.cosmos.network> and built from the files in the `/docs` directory. It is built using the following stack:

- [Docusaurus 2](#)
- Vuepress (pre v0.47)
- [Algolia DocSearch](#)

```
algolia: {
 appId: "QLS2QSP47E",
 apiKey: "067b84458bfa80c295e1d4f12c461911",
 indexName: "cosmos_network",
 contextualSearch: false,
},
```

- GitHub Pages

## Docs Build Workflow

The docs are built and deployed automatically on GitHub Pages by a [GitHub Action workflow](#). The workflow is triggered on every push to the `main` and `release/v**` branches, every time documentations or specs are modified.

### How It Works

There is a GitHub Action listening for changes in the `/docs` directory for the `main` branch and each supported version branch (e.g. `release/v0.46.x`). Any updates to files in the `/docs` directory will automatically trigger a website deployment. Under the hood, the private website repository has a `make build-docs` target consumed by a Github Action within that repository.

## How to Build the Docs Locally

Go to the `docs` directory and run the following commands:

```
cd docs
npm install
```

For starting only the current documentation, run:

```
npm start
```

It runs `pre.sh` scripts to get all the docs that are not already in the `docs/docs` folder. It also runs `post.sh` scripts to clean up the docs and remove unnecessary files when quitting.

Note, the command above only build the docs for the current versions. With the drawback that none of the redirections works. So, you'll need to go to `/main` to see the docs.

To build all the docs (including versioned documentation), run:

```
make build-docs
```

## What to do for new major SDK versions

When a new major version of the SDK is released, the following steps should be taken:

- On the `release vX.Y.Z` branch, remove the deploy action (`.github/workflows/deploy-docs.yml`), for avoiding deploying the docs from the release branches.
- On the `release vX.Y.Z` branch, update `docusaurus.config.js` and set the `lastVersion` to `current`, remove all other versions from the config.
- Each time a new version is released (on docusaurus), drop support from the oldest versions.
  - If the old version is still running vuepress (v0.45, v0.46), remove its line from `vuepress_versions`
  - If any, remove the outdated redirections from `docusaurus.config.js` and add the base version redirection (`/vX.XX`) to `/main`.

```
{
 from: ["/", "/master", "/v0.43", "/v0.44", "/v0.XX"], // here add
 the deprecated version
 to: "/main",
},
```

- Add the new version sidebar to the list of versionned sidebar and add the version to `versions.json`.
- Update the latest version (`presets[1].docs.lastVersion`) in `docusaurus.config.js`.
- Add the new version with in `presets[1].docs.versions` in `docusaurus.config.js`.

Learn more about [versioning](#) in Docusaurus.

## ADR Creation Process

1. Copy the `adr-template.md` file. Use the following filename pattern: `adr-next_number-title.md`
2. Create a draft Pull Request if you want to get an early feedback.
3. Make sure the context and a solution is clear and well documented.
4. Add an entry to a list in the [README](#) file.
5. Create a Pull Request to propose a new ADR.

## What is an ADR?

An ADR is a document to document an implementation and design that may or may not have been discussed in an RFC. While an RFC is meant to replace synchoronus communication in a distributed environment, an

ADR is meant to document an already made decision. An ADR wont come with much of a communication overhead because the discussion was recorded in an RFC or a synchronous discussion. If the consensus came from a synchoronus discussion then a short excerpt should be added to the ADR to explain the goals.

## ADR life cycle

ADR creation is an **iterative** process. Instead of having a high amount of communication overhead, an ADR is used when there is already a decision made and implementation details need to be added. The ADR should document what the collective consensus for the specific issue is and how to solve it.

1. Every ADR should start with either an RFC or discussion where consensus has been met.
2. Once consensus is met, a GitHub Pull Request (PR) is created with a new document based on the `adr-template.md`.
3. If a *proposed* ADR is merged, then it should clearly document outstanding issues either in ADR document notes or in a GitHub Issue.
4. The PR SHOULD always be merged. In the case of a faulty ADR, we still prefer to merge it with a *rejected* status. The only time the ADR SHOULD NOT be merged is if the author abandons it.
5. Merged ADRs SHOULD NOT be pruned.

## ADR status

Status has two components:

```
{CONSENSUS STATUS} {IMPLEMENTATION STATUS}
```

IMPLEMENTATION STATUS is either `Implemented` or `Not Implemented`.

### Consensus Status

```
DRAFT -> PROPOSED -> LAST CALL yyyy-mm-dd -> ACCEPTED | REJECTED -> SUPERSEDED by
ADR-xxx
```



- `DRAFT` : [optional] an ADR which is work in progress, not being ready for a general review. This is to present an early work and get an early feedback in a Draft Pull Request form.
- `PROPOSED` : an ADR covering a full solution architecture and still in the review - project stakeholders haven't reached an agreed yet.
- `LAST CALL <date for the last call>` : [optional] clear notify that we are close to accept updates. Changing a status to `LAST CALL` means that social consensus (of Cosmos SDK maintainers) has been reached and we still want to give it a time to let the community react or analyze.
- `ACCEPTED` : ADR which will represent a currently implemented or to be implemented architecture design.

- **REJECTED** : ADR can go from PROPOSED or ACCEPTED to rejected if the consensus among project stakeholders will decide so.
- **SUPERSEDED by ADR-xxx** : ADR which has been superseded by a new ADR.
- **ABANDONED** : the ADR is no longer pursued by the original authors.

## Language used in ADR

- The context/background should be written in the present tense.
  - Avoid using a first, personal form.
- 

### sidebar\_position: 1

# Architecture Decision Records (ADR)

This is a location to record all high-level architecture decisions in the Cosmos-SDK.

An Architectural Decision (**AD**) is a software design choice that addresses a functional or non-functional requirement that is architecturally significant. An Architecturally Significant Requirement (**ASR**) is a requirement that has a measurable effect on a software system's architecture and quality. An Architectural Decision Record (**ADR**) captures a single AD, such as often done when writing personal notes or meeting minutes; the collection of ADRs created and maintained in a project constitute its decision log. All these are within the topic of Architectural Knowledge Management (AKM).

You can read more about the ADR concept in this [blog post](#).

## Rationale

ADRs are intended to be the primary mechanism for proposing new feature designs and new processes, for collecting community input on an issue, and for documenting the design decisions. An ADR should provide:

- Context on the relevant goals and the current state
- Proposed changes to achieve the goals
- Summary of pros and cons
- References
- Changelog

Note the distinction between an ADR and a spec. The ADR provides the context, intuition, reasoning, and justification for a change in architecture, or for the architecture of something new. The spec is much more compressed and streamlined summary of everything as it stands today.

If recorded decisions turned out to be lacking, convene a discussion, record the new decisions here, and then modify the code to match.

## Creating new ADR

Read about the [PROCESS](#).

## Use RFC 2119 Keywords

When writing ADRs, follow the same best practices for writing RFCs. When writing RFCs, key words are used to signify the requirements in the specification. These words are often capitalized: "MUST", "MUST

NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL. They are to be interpreted as described in [RFC 2119](#).

## ADR Table of Contents

### Accepted

- [ADR 002: SDK Documentation Structure](#)
- [ADR 004: Split Denomination Keys](#)
- [ADR 006: Secret Store Replacement](#)
- [ADR 009: Evidence Module](#)
- [ADR 010: Modular AnteHandler](#)
- [ADR 019: Protocol Buffer State Encoding](#)
- [ADR 020: Protocol Buffer Transaction Encoding](#)
- [ADR 021: Protocol Buffer Query Encoding](#)
- [ADR 023: Protocol Buffer Naming and Versioning](#)
- [ADR 024: Coin Metadata](#)
- [ADR 029: Fee Grant Module](#)
- [ADR 030: Message Authorization Module](#)
- [ADR 031: Protobuf Msg Services](#)
- [ADR 046: Module Params](#)
- [ADR 055: ORM](#)
- [ADR 058: Auto-Generated CLI](#)
- [ADR 060: ABCI 1.0 \(Phase I\)](#)
- [ADR 061: Liquid Staking](#)

### Proposed

- [ADR 003: Dynamic Capability Store](#)
- [ADR 011: Generalize Genesis Accounts](#)
- [ADR 012: State Accessors](#)
- [ADR 013: Metrics](#)
- [ADR 016: Validator Consensus Key Rotation](#)
- [ADR 017: Historical Header Module](#)
- [ADR 018: Extendable Voting Periods](#)
- [ADR 022: Custom baseapp panic handling](#)
- [ADR 027: Deterministic Protobuf Serialization](#)
- [ADR 028: Public Key Addresses](#)
- [ADR 032: Typed Events](#)
- [ADR 033: Inter-module RPC](#)
- [ADR 035: Rosetta API Support](#)
- [ADR 037: Governance Split Votes](#)
- [ADR 038: State Listening](#)
- [ADR 039: Epoched Staking](#)
- [ADR 040: Storage and SMT State Commitments](#)
- [ADR 054: Semver Compatible SDK Modules](#)
- [ADR 057: App Wiring](#)
- [ADR 059: Test Scopes](#)
- [ADR 062: Collections State Layer](#)
- [ADR 063: Core Module API](#)
- [ADR 065: Store v2](#)
- [ADR 067: Simulator v2](#)

## Draft

- [ADR 044: Guidelines for Updating Protobuf Definitions](#)
- [ADR 047: Extend Upgrade Plan](#)
- [ADR 053: Go Module Refactoring](#)

# ADR 002: SDK Documentation Structure

## Context

There is a need for a scalable structure of the Cosmos SDK documentation. Current documentation includes a lot of non-related Cosmos SDK material, is difficult to maintain and hard to follow as a user.

Ideally, we would have:

- All docs related to dev frameworks or tools live in their respective github repos (sdk repo would contain sdk docs, hub repo would contain hub docs, lotion repo would contain lotion docs, etc.)
- All other docs (faqs, whitepaper, high-level material about Cosmos) would live on the website.

## Decision

Re-structure the `/docs` folder of the Cosmos SDK github repo as follows:

```
docs/
├── README
├── intro/
├── concepts/
│ ├── baseapp
│ ├── types
│ ├── store
│ ├── server
│ ├── modules/
│ │ ├── keeper
│ │ ├── handler
│ │ ├── cli
│ ├── gas
│ └── commands
├── clients/
│ ├── lite/
│ └── service-providers
├── modules/
└── spec/
└── translations/
└── architecture/
```

The files in each sub-folders do not matter and will likely change. What matters is the sectioning:

- `README` : Landing page of the docs.
- `intro` : Introductory material. Goal is to have a short explainer of the Cosmos SDK and then channel people to the resource they need. The [Cosmos SDK tutorial](#) will be highlighted, as well as the `godocs`.

- `concepts` : Contains high-level explanations of the abstractions of the Cosmos SDK. It does not contain specific code implementation and does not need to be updated often. **It is not an API specification of the interfaces.** API spec is the `godoc`.
- `clients` : Contains specs and info about the various Cosmos SDK clients.
- `spec` : Contains specs of modules, and others.
- `modules` : Contains links to `godocs` and the spec of the modules.
- `architecture` : Contains architecture-related docs like the present one.
- `translations` : Contains different translations of the documentation.

Website docs sidebar will only include the following sections:

- `README`
- `intro`
- `concepts`
- `clients`

`architecture` need not be displayed on the website.

## Status

Accepted

## Consequences

### Positive

- Much clearer organisation of the Cosmos SDK docs.
- The `/docs` folder now only contains Cosmos SDK and gaia related material. Later, it will only contain Cosmos SDK related material.
- Developers only have to update `/docs` folder when they open a PR (and not `/examples` for example).
- Easier for developers to find what they need to update in the docs thanks to reworked architecture.
- Cleaner vuepress build for website docs.
- Will help build an executable doc (cf <https://github.com/cosmos/cosmos-sdk/issues/2611>)

### Neutral

- We need to move a bunch of deprecated stuff to `/attic` folder.
- We need to integrate content in `docs/sdk/docs/core` in `concepts`.
- We need to move all the content that currently lives in `docs` and does not fit in new structure (like `lotion`, intro material, whitepaper) to the website repository.
- Update `DOCS_README.md`

## References

- <https://github.com/cosmos/cosmos-sdk/issues/1460>
- <https://github.com/cosmos/cosmos-sdk/pull/2695>
- <https://github.com/cosmos/cosmos-sdk/issues/2611>

## ADR 3: Dynamic Capability Store

## Changelog

- 12 December 2019: Initial version
- 02 April 2020: Memory Store Revisions

## Context

Full implementation of the [IBC specification](#) requires the ability to create and authenticate object-capability keys at runtime (i.e., during transaction execution), as described in [ICS 5](#). In the IBC specification, capability keys are created for each newly initialised port & channel, and are used to authenticate future usage of the port or channel. Since channels and potentially ports can be initialised during transaction execution, the state machine must be able to create object-capability keys at this time.

At present, the Cosmos SDK does not have the ability to do this. Object-capability keys are currently pointers (memory addresses) of `StoreKey` structs created at application initialisation in `app.go` ([example](#)) and passed to Keepers as fixed arguments ([example](#)). Keepers cannot create or store capability keys during transaction execution — although they could call `NewKVStoreKey` and take the memory address of the returned struct, storing this in the Merklised store would result in a consensus fault, since the memory address will be different on each machine (this is intentional — were this not the case, the keys would be predictable and couldn't serve as object capabilities).

Keepers need a way to keep a private map of store keys which can be altered during transaction execution, along with a suitable mechanism for regenerating the unique memory addresses (capability keys) in this map whenever the application is started or restarted, along with a mechanism to revert capability creation on tx failure. This ADR proposes such an interface & mechanism.

## Decision

The Cosmos SDK will include a new `CapabilityKeeper` abstraction, which is responsible for provisioning, tracking, and authenticating capabilities at runtime. During application initialisation in `app.go`, the `CapabilityKeeper` will be hooked up to modules through unique function references (by calling `ScopeToModule`, defined below) so that it can identify the calling module when later invoked.

When the initial state is loaded from disk, the `CapabilityKeeper`'s `Initialise` function will create new capability keys for all previously allocated capability identifiers (allocated during execution of past transactions and assigned to particular modes), and keep them in a memory-only store while the chain is running.

The `CapabilityKeeper` will include a persistent `KVStore`, a `MemoryStore`, and an in-memory map. The persistent `KVStore` tracks which capability is owned by which modules. The `MemoryStore` stores a forward mapping that map from module name, capability tuples to capability names and a reverse mapping that map from module name, capability name to the capability index. Since we cannot marshal the capability into a `KVStore` and unmarshal without changing the memory location of the capability, the reverse mapping in the `KVStore` will simply map to an index. This index can then be used as a key in the ephemeral go-map to retrieve the capability at the original memory location.

The `CapabilityKeeper` will define the following types & functions:

The `Capability` is similar to `StoreKey`, but has a globally unique `Index()` instead of a name. A `String()` method is provided for debugging.

A `Capability` is simply a struct, the address of which is taken for the actual capability.

```
type Capability struct {
 index uint64
}
```

A `CapabilityKeeper` contains a persistent store key, memory store key, and mapping of allocated module names.

```
type CapabilityKeeper struct {
 persistentKey StoreKey
 memKey StoreKey
 capMap map[uint64]*Capability
 moduleNameNames map[string]interface{}
 sealed bool
}
```

The `CapabilityKeeper` provides the ability to create *scoped* sub-keepers which are tied to a particular module name. These `ScopedCapabilityKeeper`s must be created at application initialisation and passed to modules, which can then use them to claim capabilities they receive and retrieve capabilities which they own by name, in addition to creating new capabilities & authenticating capabilities passed by other modules.

```
type ScopedCapabilityKeeper struct {
 persistentKey StoreKey
 memKey StoreKey
 capMap map[uint64]*Capability
 moduleName string
}
```

`ScopeToModule` is used to create a scoped sub-keeper with a particular name, which must be unique. It MUST be called before `InitialiseAndSeal`.

```
func (ck CapabilityKeeper) ScopeToModule(moduleName string) ScopedCapabilityKeeper {
 if k.sealed {
 panic("cannot scope to module via a sealed capability keeper")
 }

 if _, ok := k.scopedModules[moduleName]; ok {
 panic(fmt.Sprintf("cannot create multiple scoped keepers for the same module name: %s", moduleName))
 }

 k.scopedModules[moduleName] = struct{}{}

 return ScopedKeeper{
 cdc: k.cdc,
 storeKey: k.storeKey,
 memKey: k.memKey,
 capMap: k.capMap,
 module: moduleName,
 }
}
```

```
 }
}
```

`InitialiseAndSeal` MUST be called exactly once, after loading the initial state and creating all necessary `ScopedCapabilityKeeper`s, in order to populate the memory store with newly-created capability keys in accordance with the keys previously claimed by particular modules and prevent the creation of any new `ScopedCapabilityKeeper`s.

```
func (ck CapabilityKeeper) InitialiseAndSeal(ctx Context) {
 if cksealed {
 panic("capability keeper is sealed")
 }

 persistentStore := ctx.KVStore(ck.persistentKey)
 map := ctx.KVStore(ck.memKey)

 // initialise memory store for all names in persistent store
 for index, value := range persistentStore.Iter() {
 capability = &CapabilityKey{index: index}

 for moduleAndCapability := range value {
 moduleName, capabilityName := moduleAndCapability.Split("/")
 memStore.Set(moduleName + "/fwd/" + capability, capabilityName)
 memStore.Set(moduleName + "/rev/" + capabilityName, index)

 ck.capMap[index] = capability
 }
 }

 cksealed = true
}
```

`NewCapability` can be called by any module to create a new unique, unforgeable object-capability reference. The newly created capability is automatically persisted; the calling module need not call `ClaimCapability`.

```
func (sck ScopedCapabilityKeeper) NewCapability(ctx Context, name string)
(Capability, error) {
 // check name not taken in memory store
 if capStore.Get("rev/" + name) != nil {
 return nil, errors.New("name already taken")
 }

 // fetch the current index
 index := persistentStore.Get("index")

 // create a new capability
 capability := &CapabilityKey{index: index}

 // set persistent store
```

```

persistentStore.Set(index, Set.singleton(sck.moduleName + "/" + name))

// update the index
index++
persistentStore.Set("index", index)

// set forward mapping in memory store from capability to name
memStore.Set(sck.moduleName + "/fwd/" + capability, name)

// set reverse mapping in memory store from name to index
memStore.Set(sck.moduleName + "/rev/" + name, index)

// set the in-memory mapping from index to capability pointer
capMap[index] = capability

// return the newly created capability
return capability
}

```

`AuthenticateCapability` can be called by any module to check that a capability does in fact correspond to a particular name (the name can be untrusted user input) with which the calling module previously associated it.

```

func (sck ScopedCapabilityKeeper) AuthenticateCapability(name string, capability
Capability) bool {
 // return whether forward mapping in memory store matches name
 return memStore.Get(sck.moduleName + "/fwd/" + capability) === name
}

```

`ClaimCapability` allows a module to claim a capability key which it has received from another module so that future `GetCapability` calls will succeed.

`ClaimCapability` MUST be called if a module which receives a capability wishes to access it by name in the future. Capabilities are multi-owner, so if multiple modules have a single `Capability` reference, they will all own it.

```

func (sck ScopedCapabilityKeeper) ClaimCapability(ctx Context, capability
Capability, name string) error {
 persistentStore := ctx.KVStore(sck.persistentKey)

 // set forward mapping in memory store from capability to name
 memStore.Set(sck.moduleName + "/fwd/" + capability, name)

 // set reverse mapping in memory store from name to capability
 memStore.Set(sck.moduleName + "/rev/" + name, capability)

 // update owner set in persistent store
 owners := persistentStore.Get(capability.Index())
 owners.add(sck.moduleName + "/" + name)
}

```

```
 persistentStore.Set(capability.Index(), owners)
}
```

`GetCapability` allows a module to fetch a capability which it has previously claimed by name. The module is not allowed to retrieve capabilities which it does not own.

```
func (sck ScopedCapabilityKeeper) GetCapability(ctx Context, name string)
(Capability, error) {
 // fetch the index of capability using reverse mapping in memstore
 index := memStore.Get(sck.moduleName + "/rev/" + name)

 // fetch capability from go-map using index
 capability := capMap[index]

 // return the capability
 return capability
}
```

`ReleaseCapability` allows a module to release a capability which it had previously claimed. If no more owners exist, the capability will be deleted globally.

```
func (sck ScopedCapabilityKeeper) ReleaseCapability(ctx Context, capability
Capability) err {
 persistentStore := ctx.KVStore(sck.persistentKey)

 name := capStore.Get(sck.moduleName + "/fwd/" + capability)
 if name == nil {
 return error("capability not owned by module")
 }

 // delete forward mapping in memory store
 memoryStore.Delete(sck.moduleName + "/fwd/" + capability, name)

 // delete reverse mapping in memory store
 memoryStore.Delete(sck.moduleName + "/rev/" + name, capability)

 // update owner set in persistent store
 owners := persistentStore.Get(capability.Index())
 owners.remove(sck.moduleName + "/" + name)
 if owners.size() > 0 {
 // there are still other owners, keep the capability around
 persistentStore.Set(capability.Index(), owners)
 } else {
 // no more owners, delete the capability
 persistentStore.Delete(capability.Index())
 delete(capMap[capability.Index()])
 }
}
```

## Usage patterns

### Initialisation

Any modules which use dynamic capabilities must be provided a `ScopedCapabilityKeeper` in `app.go`:

```
ck := NewCapabilityKeeper(persistentKey, memoryKey)
mod1Keeper := NewMod1Keeper(ck.ScopeToModule("mod1"), ...)
mod2Keeper := NewMod2Keeper(ck.ScopeToModule("mod2"), ...)

// other initialisation logic ...

// load initial state...

ck.InitialiseAndSeal(initialContext)
```

### Creating, passing, claiming and using capabilities

Consider the case where `mod1` wants to create a capability, associate it with a resource (e.g. an IBC channel) by name, then pass it to `mod2` which will use it later:

Module 1 would have the following code:

```
capability := scopedCapabilityKeeper.NewCapability(ctx, "resourceABC")
mod2Keeper.SomeFunction(ctx, capability, args...)
```

`SomeFunction`, running in module 2, could then claim the capability:

```
func (k Mod2Keeper) SomeFunction(ctx Context, capability Capability) {
 k.sck.ClaimCapability(ctx, capability, "resourceABC")
 // other logic...
}
```

Later on, module 2 can retrieve that capability by name and pass it to module 1, which will authenticate it against the resource:

```
func (k Mod2Keeper) SomeOtherFunction(ctx Context, name string) {
 capability := k.sck.GetCapability(ctx, name)
 mod1.UseResource(ctx, capability, "resourceABC")
}
```

Module 1 will then check that this capability key is authenticated to use the resource before allowing module 2 to use it:

```
func (k Mod1Keeper) UseResource(ctx Context, capability Capability, resource string)
{
 if !k.sck.AuthenticateCapability(name, capability) {
 return errors.New("unauthenticated")
 }
}
```

```
// do something with the resource
}
```

If module 2 passed the capability key to module 3, module 3 could then claim it and call module 1 just like module 2 did (in which case module 1, module 2, and module 3 would all be able to use this capability).

## Status

Proposed.

## Consequences

### Positive

- Dynamic capability support.
- Allows CapabilityKeeper to return same capability pointer from go-map while reverting any writes to the persistent `KVStore` and in-memory `MemoryStore` on tx failure.

### Negative

- Requires an additional keeper.
- Some overlap with existing `StoreKey` system (in the future they could be combined, since this is a superset functionality-wise).
- Requires an extra level of indirection in the reverse mapping, since `MemoryStore` must map to index which must then be used as key in a go map to retrieve the actual capability

### Neutral

(none known)

## References

- [Original discussion](#)

# ADR 004: Split Denomination Keys

## Changelog

- 2020-01-08: Initial version
- 2020-01-09: Alterations to handle vesting accounts
- 2020-01-14: Updates from review feedback
- 2020-01-30: Updates from implementation

## Glossary

- denom / denomination key -- unique token identifier.

## Context

With permissionless IBC, anyone will be able to send arbitrary denominations to any other account.

Currently, all non-zero balances are stored along with the account in an `sdk.Coins` struct, which creates a potential denial-of-service concern, as too many denominations will become expensive to load & store each time the account is modified. See issues [5467](#) and [4982](#) for additional context.

Simply rejecting incoming deposits after a denomination count limit doesn't work, since it opens up a griefing vector: someone could send a user lots of nonsensical coins over IBC, and then prevent the user from receiving real denominations (such as staking rewards).

## Decision

Balances shall be stored per-account & per-denomination under a denomination- and account-unique key, thus enabling O(1) read & write access to the balance of a particular account in a particular denomination.

### Account interface (x/auth)

`GetCoins()` and `SetCoins()` will be removed from the account interface, since coin balances will now be stored in & managed by the bank module.

The vesting account interface will replace `SpendableCoins` in favor of `LockedCoins` which does not require the account balance anymore. In addition, `TrackDelegation()` will now accept the account balance of all tokens denominated in the vesting balance instead of loading the entire account balance.

Vesting accounts will continue to store original vesting, delegated free, and delegated vesting coins (which is safe since these cannot contain arbitrary denominations).

### Bank keeper (x/bank)

The following APIs will be added to the `x/bank` keeper:

- `GetAllBalances(ctx Context, addr AccAddress) Coins`
- `GetBalance(ctx Context, addr AccAddress, denom string) Coin`
- `SetBalance(ctx Context, addr AccAddress, coin Coin)`
- `LockedCoins(ctx Context, addr AccAddress) Coins`
- `SpendableCoins(ctx Context, addr AccAddress) Coins`

Additional APIs may be added to facilitate iteration and auxiliary functionality not essential to core functionality or persistence.

Balances will be stored first by the address, then by the denomination (the reverse is also possible, but retrieval of all balances for a single account is presumed to be more frequent):

```
var BalancesPrefix = []byte("balances")

func (k Keeper) SetBalance(ctx Context, addr AccAddress, balance Coin) error {
 if !balance.IsValid() {
 return err
 }

 store := ctx.KVStore(k.storeKey)
 balancesStore := prefix.NewStore(store, BalancesPrefix)
 accountStore := prefix.NewStore(balancesStore, addr.Bytes())

 bz := Marshal(balance)
 accountStore.Set([]byte(balance.Denom), bz)
```

```
 return nil
}
```

This will result in the balances being indexed by the byte representation of `balances/{address}/{denom}`.

`DelegateCoins()` and `UndelegateCoins()` will be altered to only load each individual account balance by denomination found in the (un)delegation amount. As a result, any mutations to the account balance by will made by denomination.

`SubtractCoins()` and `AddCoins()` will be altered to read & write the balances directly instead of calling `GetCoins()` / `SetCoins()` (which no longer exist).

`trackDelegation()` and `trackUndelegation()` will be altered to no longer update account balances.

External APIs will need to scan all balances under an account to retain backwards-compatibility. It is advised that these APIs use `GetBalance` and `SetBalance` instead of `GetAllBalances` when possible as to not load the entire account balance.

## Supply module

The supply module, in order to implement the total supply invariant, will now need to scan all accounts & call `GetAllBalances` using the `x/bank` Keeper, then sum the balances and check that they match the expected total supply.

## Status

Accepted.

## Consequences

### Positive

- O(1) reads & writes of balances (with respect to the number of denominations for which an account has non-zero balances). Note, this does not relate to the actual I/O cost, rather the total number of direct reads needed.

### Negative

- Slightly less efficient reads/writes when reading & writing all balances of a single account in a transaction.

### Neutral

None in particular.

## References

- Ref: <https://github.com/cosmos/cosmos-sdk/issues/4982>
- Ref: <https://github.com/cosmos/cosmos-sdk/issues/5467>
- Ref: <https://github.com/cosmos/cosmos-sdk/issues/5492>

# ADR 006: Secret Store Replacement

## Changelog

- July 29th, 2019: Initial draft
- September 11th, 2019: Work has started
- November 4th: Cosmos SDK changes merged in
- November 18th: Gaia changes merged in

## Context

Currently, a Cosmos SDK application's CLI directory stores key material and metadata in a plain text database in the user's home directory. Key material is encrypted by a passphrase, protected by bcrypt hashing algorithm. Metadata (e.g. addresses, public keys, key storage details) is available in plain text.

This is not desirable for a number of reasons. Perhaps the biggest reason is insufficient security protection of key material and metadata. Leaking the plain text allows an attacker to surveil what keys a given computer controls via a number of techniques, like compromised dependencies without any privilege execution. This could be followed by a more targeted attack on a particular user/computer.

All modern desktop computers OS (Ubuntu, Debian, MacOS, Windows) provide a built-in secret store that is designed to allow applications to store information that is isolated from all other applications and requires passphrase entry to access the data.

We are seeking solution that provides a common abstraction layer to the many different backends and reasonable fallback for minimal platforms that don't provide a native secret store.

## Decision

We recommend replacing the current Keybase backend based on LevelDB with [Keyring](#) by 99 designs. This application is designed to provide a common abstraction and uniform interface between many secret stores and is used by AWS Vault application by 99-designs application.

This appears to fulfill the requirement of protecting both key material and metadata from rogue software on a user's machine.

## Status

Accepted

## Consequences

### Positive

Increased safety for users.

### Negative

Users must manually migrate.

Testing against all supported backends is difficult.

Running tests locally on a Mac require numerous repetitive password entries.

### Neutral

{neutral consequences}

## References

- #4754 Switch secret store to the keyring secret store (original PR by @poldsam) [**CLOSED**]
- #5029 Add support for github.com/99designs/keyring-backed keybases [**MERGED**]
- #5097 Add keys migrate command [**MERGED**]
- #5180 Drop on-disk keybase in favor of keyring [*PENDING REVIEW*]
- cosmos/gaia#164 Drop on-disk keybase in favor of keyring (gaia's changes) [*PENDING REVIEW*]

# ADR 007: Specialization Groups

## Changelog

- 2019 Jul 31: Initial Draft

## Context

This idea was first conceived of in order to fulfill the use case of the creation of a decentralized Computer Emergency Response Team (dCERT), whose members would be elected by a governing community and would fulfill the role of coordinating the community under emergency situations. This thinking can be further abstracted into the conception of "blockchain specialization groups".

The creation of these groups are the beginning of specialization capabilities within a wider blockchain community which could be used to enable a certain level of delegated responsibilities. Examples of specialization which could be beneficial to a blockchain community include: code auditing, emergency response, code development etc. This type of community organization paves the way for individual stakeholders to delegate votes by issue type, if in the future governance proposals include a field for issue type.

## Decision

A specialization group can be broadly broken down into the following functions (herein containing examples):

- Membership Admittance
- Membership Acceptance
- Membership Revocation
  - (probably) Without Penalty
    - member steps down (self-Revocation)
    - replaced by new member from governance
  - (probably) With Penalty
    - due to breach of soft-agreement (determined through governance)
    - due to breach of hard-agreement (determined by code)
- Execution of Duties
  - Special transactions which only execute for members of a specialization group (for example, dCERT members voting to turn off transaction routes in an emergency scenario)
- Compensation
  - Group compensation (further distribution decided by the specialization group)

- Individual compensation for all constituents of a group from the greater community

Membership admittance to a specialization group could take place over a wide variety of mechanisms. The most obvious example is through a general vote among the entire community, however in certain systems a community may want to allow the members already in a specialization group to internally elect new members, or maybe the community may assign a permission to a particular specialization group to appoint members to other 3rd party groups. The sky is really the limit as to how membership admittance can be structured. We attempt to capture some of these possibilities in a common interface dubbed the `Electionator`. For its initial implementation as a part of this ADR we recommend that the general election abstraction (`Electionator`) is provided as well as a basic implementation of that abstraction which allows for a continuous election of members of a specialization group.

```
// The Electionator abstraction covers the concept space for
// a wide variety of election kinds.
type Electionator interface {

 // is the election object accepting votes.
 Active() bool

 // functionality to execute for when a vote is cast in this election, here
 // the vote field is anticipated to be marshalled into a vote type used
 // by an election.
 //
 // NOTE There are no explicit ids here. Just votes which pertain specifically
 // to one electionator. Anyone can create and send a vote to the electionator
 item
 // which will presumably attempt to marshal those bytes into a particular struct
 // and apply the vote information in some arbitrary way. There can be multiple
 // Electionators within the Cosmos-Hub for multiple specialization groups, votes
 // would need to be routed to the Electionator upstream of here.
 Vote(addr sdk.AccAddress, vote []byte)

 // here lies all functionality to authenticate and execute changes for
 // when a member accepts being elected
 AcceptElection(sdk.AccAddress)

 // Register a revoker object
 RegisterRevoker(Revoker)

 // No more revokers may be registered after this function is called
 SealRevokers()

 // register hooks to call when an election actions occur
 RegisterHooks(ElectionatorHooks)

 // query for the current winner(s) of this election based on arbitrary
 // election ruleset
 QueryElected() []sdk.AccAddress

 // query metadata for an address in the election this
 // could include for example position that an address
```

```

// is being elected for within a group
//
// this metadata may be directly related to
// voting information and/or privileges enabled
// to members within a group.
QueryMetadata(sdk.AccAddress) []byte
}

// ElectionatorHooks, once registered with an Electionator,
// trigger execution of relevant interface functions when
// Electionator events occur.
type ElectionatorHooks interface {
 AfterVoteCast(addr sdk.AccAddress, vote []byte)
 AfterMemberAccepted(addr sdk.AccAddress)
 AfterMemberRevoked(addr sdk.AccAddress, cause []byte)
}
}

// Revoker defines the function required for a membership revocation rule-set
// used by a specialization group. This could be used to create self revoking,
// and evidence based revoking, etc. Revokers types may be created and
// reused for different election types.
//
// When revoking the "cause" bytes may be arbitrarily marshalled into evidence,
// memos, etc.
type Revoker interface {
 RevokeName() string // identifier for this revoker type
 RevokeMember(addr sdk.AccAddress, cause []byte) error
}

```

Certain level of commonality likely exists between the existing code within `x/governance` and required functionality of elections. This common functionality should be abstracted during implementation. Similarly for each vote implementation client CLI/REST functionality should be abstracted to be reused for multiple elections.

The specialization group abstraction firstly extends the `Electionator` but also further defines traits of the group.

```

type SpecializationGroup interface {
 Electionator
 GetName() string
 GetDescription() string

 // general soft contract the group is expected
 // to fulfill with the greater community
 GetContract() string

 // messages which can be executed by the members of the group
 Handler(ctx sdk.Context, msg sdk.Msg) sdk.Result

 // logic to be executed at endblock, this may for instance
 // include payment of a stipend to the group members
}

```

```
// for participation in the security group.
EndBlocker(ctx sdk.Context)
}
```

## Status

*Proposed*

## Consequences

### Positive

- increases specialization capabilities of a blockchain
- improve abstractions in `x/gov/` such that they can be used with specialization groups

### Negative

- could be used to increase centralization within a community

### Neutral

## References

- [dCERT ADR](#)

# ADR 008: Decentralized Computer Emergency Response Team (dCERT) Group

## Changelog

- 2019 Jul 31: Initial Draft

## Context

In order to reduce the number of parties involved with handling sensitive information in an emergency scenario, we propose the creation of a specialization group named The Decentralized Computer Emergency Response Team (dCERT). Initially this group's role is intended to serve as coordinators between various actors within a blockchain community such as validators, bug-hunters, and developers. During a time of crisis, the dCERT group would aggregate and relay input from a variety of stakeholders to the developers who are actively devising a patch to the software, this way sensitive information does not need to be publicly disclosed while some input from the community can still be gained.

Additionally, a special privilege is proposed for the dCERT group: the capacity to "circuit-break" (aka. temporarily disable) a particular message path. Note that this privilege should be enabled/disabled globally with a governance parameter such that this privilege could start disabled and later be enabled through a parameter change proposal, once a dCERT group has been established.

In the future it is foreseeable that the community may wish to expand the roles of dCERT with further responsibilities such as the capacity to "pre-approve" a security update on behalf of the community prior to a full community wide vote whereby the sensitive information would be revealed prior to a vulnerability being patched on the live network.

## Decision

The dCERT group is proposed to include an implementation of a `SpecializationGroup` as defined in [ADR 007](#). This will include the implementation of:

- continuous voting
- slashing due to breach of soft contract
- revoking a member due to breach of soft contract
- emergency disband of the entire dCERT group (ex. for colluding maliciously)
- compensation stipend from the community pool or other means decided by governance

This system necessitates the following new parameters:

- blockly stipend allowance per dCERT member
- maximum number of dCERT members
- required staked slashable tokens for each dCERT member
- quorum for suspending a particular member
- proposal wager for disbanding the dCERT group
- stabilization period for dCERT member transition
- circuit break dCERT privileges enabled

These parameters are expected to be implemented through the param keeper such that governance may change them at any given point.

### Continuous Voting Electionator

An `Electionator` object is to be implemented as continuous voting and with the following specifications:

- All delegation addresses may submit votes at any point which updates their preferred representation on the dCERT group.
- Preferred representation may be arbitrarily split between addresses (ex. 50% to John, 25% to Sally, 25% to Carol)
- In order for a new member to be added to the dCERT group they must send a transaction accepting their admission at which point the validity of their admission is to be confirmed.
  - A sequence number is assigned when a member is added to dCERT group. If a member leaves the dCERT group and then enters back, a new sequence number is assigned.
- Addresses which control the greatest amount of preferred-representation are eligible to join the dCERT group (up the *maximum number of dCERT members*). If the dCERT group is already full and new member is admitted, the existing dCERT member with the lowest amount of votes is kicked from the dCERT group.
  - In the split situation where the dCERT group is full but a vying candidate has the same amount of vote as an existing dCERT member, the existing member should maintain its position.
  - In the split situation where somebody must be kicked out but the two addresses with the smallest number of votes have the same number of votes, the address with the smallest sequence number maintains its position.
- A stabilization period can be optionally included to reduce the "flip-flopping" of the dCERT membership tail members. If a stabilization period is provided which is greater than 0, when members are kicked due to insufficient support, a queue entry is created which documents which member is to replace which other member. While this entry is in the queue, no new entries to kick that same dCERT member can be made. When the entry matures at the duration of the stabilization period, the new member is instantiated, and old member kicked.

## **Staking/Slashing**

All members of the dCERT group must stake tokens *specifically* to maintain eligibility as a dCERT member. These tokens can be staked directly by the vying dCERT member or out of the good will of a 3rd party (who shall gain no on-chain benefits for doing so). This staking mechanism should use the existing global unbonding time of tokens staked for network validator security. A dCERT member can *only* be a member if it has the required tokens staked under this mechanism. If those tokens are unbonded then the dCERT member must be automatically kicked from the group.

Slashing of a particular dCERT member due to soft-contract breach should be performed by governance on a per member basis based on the magnitude of the breach. The process flow is anticipated to be that a dCERT member is suspended by the dCERT group prior to being slashed by governance.

Membership suspension by the dCERT group takes place through a voting procedure by the dCERT group members. After this suspension has taken place, a governance proposal to slash the dCERT member must be submitted, if the proposal is not approved by the time the rescinding member has completed unbonding their tokens, then the tokens are no longer staked and unable to be slashed.

Additionally in the case of an emergency situation of a colluding and malicious dCERT group, the community needs the capability to disband the entire dCERT group and likely fully slash them. This could be achieved though a special new proposal type (implemented as a general governance proposal) which would halt the functionality of the dCERT group until the proposal was concluded. This special proposal type would likely need to also have a fairly large wager which could be slashed if the proposal creator was malicious. The reason a large wager should be required is because as soon as the proposal is made, the capability of the dCERT group to halt message routes is put on temporarily suspended, meaning that a malicious actor who created such a proposal could then potentially exploit a bug during this period of time, with no dCERT group capable of shutting down the exploitable message routes.

## **dCERT membership transactions**

Active dCERT members

- change of the description of the dCERT group
- circuit break a message route
- vote to suspend a dCERT member.

Here circuit-breaking refers to the capability to disable a groups of messages, This could for instance mean: "disable all staking-delegation messages", or "disable all distribution messages". This could be accomplished by verifying that the message route has not been "circuit-broken" at CheckTx time (in baseapp/baseapp.go ).

"unbreaking" a circuit is anticipated only to occur during a hard fork upgrade meaning that no capability to unbreak a message route on a live chain is required.

Note also, that if there was a problem with governance voting (for instance a capability to vote many times) then governance would be broken and should be halted with this mechanism, it would be then up to the validator set to coordinate and hard-fork upgrade to a patched version of the software where governance is re-enabled (and fixed). If the dCERT group abuses this privilege they should all be severely slashed.

## **Status**

*Proposed*

## Consequences

### Positive

- Potential to reduce the number of parties to coordinate with during an emergency
- Reduction in possibility of disclosing sensitive information to malicious parties

### Negative

- Centralization risks

### Neutral

## References

[Specialization Groups ADR](#)

# ADR 009: Evidence Module

## Changelog

- 2019 July 31: Initial draft
- 2019 October 24: Initial implementation

## Status

Accepted

## Context

In order to support building highly secure, robust and interoperable blockchain applications, it is vital for the Cosmos SDK to expose a mechanism in which arbitrary evidence can be submitted, evaluated and verified resulting in some agreed upon penalty for any misbehavior committed by a validator, such as equivocation (double-voting), signing when unbonded, signing an incorrect state transition (in the future), etc. Furthermore, such a mechanism is paramount for any [IBC](#) or cross-chain validation protocol implementation in order to support the ability for any misbehavior to be relayed back from a collateralized chain to a primary chain so that the equivocating validator(s) can be slashed.

## Decision

We will implement an evidence module in the Cosmos SDK supporting the following functionality:

- Provide developers with the abstractions and interfaces necessary to define custom evidence messages, message handlers, and methods to slash and penalize accordingly for misbehavior.
- Support the ability to route evidence messages to handlers in any module to determine the validity of submitted misbehavior.
- Support the ability, through governance, to modify slashing penalties of any evidence type.
- Querier implementation to support querying params, evidence types, params, and all submitted valid misbehavior.

## Types

First, we define the `Evidence` interface type. The `x/evidence` module may implement its own types that can be used by many chains (e.g. `CounterFactualEvidence`). In addition, other modules may implement their own `Evidence` types in a similar manner in which governance is extensible. It is important to note any concrete type implementing the `Evidence` interface may include arbitrary fields such as an infraction time. We want the `Evidence` type to remain as flexible as possible.

When submitting evidence to the `x/evidence` module, the concrete type must provide the validator's consensus address, which should be known by the `x/slashing` module (assuming the infraction is valid), the height at which the infraction occurred and the validator's power at same height in which the infraction occurred.

```
type Evidence interface {
 Route() string
 Type() string
 String() string
 Hash() HexBytes
 ValidateBasic() error

 // The consensus address of the malicious validator at time of infraction
 GetConsensusAddress() ConsAddress

 // Height at which the infraction occurred
 GetHeight() int64

 // The total power of the malicious validator at time of infraction
 GetValidatorPower() int64

 // The total validator set power at time of infraction
 GetTotalPower() int64
}
```

## Routing & Handling

Each `Evidence` type must map to a specific unique route and be registered with the `x/evidence` module. It accomplishes this through the `Router` implementation.

```
type Router interface {
 AddRoute(r string, h Handler) Router
 HasRoute(r string) bool
 GetRoute(path string) Handler
 Seal()
}
```

Upon successful routing through the `x/evidence` module, the `Evidence` type is passed through a `Handler`. This `Handler` is responsible for executing all corresponding business logic necessary for verifying the evidence as valid. In addition, the `Handler` may execute any necessary slashing and potential jailing. Since slashing fractions will typically result from some form of static functions, allowing the `Handler` to do this provides the greatest flexibility. An example could be `k * evidence.GetValidatorPower()` where `k` is an on-chain parameter controlled by governance. The `Evidence` type should provide all the

external information necessary in order for the `Handler` to make the necessary state transitions. If no error is returned, the `Evidence` is considered valid.

```
type Handler func(Context, Evidence) error
```

## Submission

`Evidence` is submitted through a `MsgSubmitEvidence` message type which is internally handled by the `x/evidence` module's `SubmitEvidence`.

```
type MsgSubmitEvidence struct {
 Evidence
}

func handleMsgSubmitEvidence(ctx Context, keeper Keeper, msg MsgSubmitEvidence) Result {
 if err := keeper.SubmitEvidence(ctx, msg.Evidence); err != nil {
 return err.Result()
 }

 // emit events...

 return Result{
 // ...
 }
}
```

The `x/evidence` module's `keeper` is responsible for matching the `Evidence` against the module's router and invoking the corresponding `Handler` which may include slashing and jailing the validator. Upon success, the submitted evidence is persisted.

```
func (k Keeper) SubmitEvidence(ctx Context, evidence Evidence) error {
 handler := keeper.router.GetRoute(evidence.Route())
 if err := handler(ctx, evidence); err != nil {
 return ErrInvalidEvidence(keeper.codespace, err)
 }

 keeper.setEvidence(ctx, evidence)
 return nil
}
```

## Genesis

Finally, we need to represent the genesis state of the `x/evidence` module. The module only needs a list of all submitted valid infractions and any necessary params for which the module needs in order to handle submitted evidence. The `x/evidence` module will naturally define and route native evidence types for which it'll most likely need slashing penalty constants for.

```
type GenesisState struct {
 Params Params
 Infractions []Evidence
}
```

## Consequences

### Positive

- Allows the state machine to process misbehavior submitted on-chain and penalize validators based on agreed upon slashing parameters.
- Allows evidence types to be defined and handled by any module. This further allows slashing and jailing to be defined by more complex mechanisms.
- Does not solely rely on Tendermint to submit evidence.

### Negative

- No easy way to introduce new evidence types through governance on a live chain due to the inability to introduce the new evidence type's corresponding handler

### Neutral

- Should we persist infractions indefinitely? Or should we rather rely on events?

## References

- [ICS](#)
- [IBC Architecture](#)
- [Tendermint Fork Accountability](#)

## ADR 010: Modular AnteHandler

### Changelog

- 2019 Aug 31: Initial draft
- 2021 Sep 14: Superseded by ADR-045

### Status

SUPERSEDED by ADR-045

### Context

The current AnteHandler design allows users to either use the default AnteHandler provided in `x/auth` or to build their own AnteHandler from scratch. Ideally AnteHandler functionality is split into multiple, modular functions that can be chained together along with custom ante-functions so that users do not have to rewrite common antehandler logic when they want to implement custom behavior.

For example, let's say a user wants to implement some custom signature verification logic. In the current codebase, the user would have to write their own Antehandler from scratch largely reimplementing much of the same code and then set their own custom, monolithic antehandler in the baseapp. Instead, we would like

to allow users to specify custom behavior when necessary and combine them with default ante-handler functionality in a way that is as modular and flexible as possible.

## Proposals

### Per-Module AnteHandler

One approach is to use the [ModuleManager](#) and have each module implement its own antehandler if it requires custom antehandler logic. The ModuleManager can then be passed in an AnteHandler order in the same way it has an order for BeginBlockers and EndBlockers. The ModuleManager returns a single AnteHandler function that will take in a tx and run each module's `AnteHandle` in the specified order. The module manager's AnteHandler is set as the baseapp's AnteHandler.

Pros:

1. Simple to implement
2. Utilizes the existing ModuleManager architecture

Cons:

1. Improves granularity but still cannot get more granular than a per-module basis. e.g. If auth's `AnteHandle` function is in charge of validating memo and signatures, users cannot swap the signature-checking functionality while keeping the rest of auth's `AnteHandle` functionality.
2. Module AnteHandler are run one after the other. There is no way for one AnteHandler to wrap or "decorate" another.

### Decorator Pattern

The [weave project](#) achieves AnteHandler modularity through the use of a decorator pattern. The interface is designed as follows:

```
// Decorator wraps a Handler to provide common functionality
// like authentication, or fee-handling, to many Handlers
type Decorator interface {
 Check(ctx Context, store KVStore, tx Tx, next Checker) (*CheckResult, error)
 Deliver(ctx Context, store KVStore, tx Tx, next Deliverer) (*DeliverResult,
error)
}
```

Each decorator works like a modularized Cosmos SDK antehandler function, but it can take in a `next` argument that may be another decorator or a Handler (which does not take in a next argument). These decorators can be chained together, one decorator being passed in as the `next` argument of the previous decorator in the chain. The chain ends in a Router which can take a tx and route to the appropriate msg handler.

A key benefit of this approach is that one Decorator can wrap its internal logic around the next Checker/Deliverer. A weave Decorator may do the following:

```
// Example Decorator's Deliver function
func (example Decorator) Deliver(ctx Context, store KVStore, tx Tx, next Deliverer)
{
 // Do some pre-processing logic
```

```

 res, err := next.Deliver(ctx, store, tx)

 // Do some post-processing logic given the result and error
}

```

Pros:

1. Weave Decorators can wrap over the next decorator/handler in the chain. The ability to both pre-process and post-process may be useful in certain settings.
2. Provides a nested modular structure that isn't possible in the solution above, while also allowing for a linear one-after-the-other structure like the solution above.

Cons:

1. It is hard to understand at first glance the state updates that would occur after a Decorator runs given the `ctx`, `store`, and `tx`. A Decorator can have an arbitrary number of nested Decorators being called within its function body, each possibly doing some pre- and post-processing before calling the next decorator on the chain. Thus to understand what a Decorator is doing, one must also understand what every other decorator further along the chain is also doing. This can get quite complicated to understand. A linear, one-after-the-other approach while less powerful, may be much easier to reason about.

## **Chained Micro-Functions**

The benefit of Weave's approach is that the Decorators can be very concise, which when chained together allows for maximum customizability. However, the nested structure can get quite complex and thus hard to reason about.

Another approach is to split the AnteHandler functionality into tightly scoped "micro-functions", while preserving the one-after-the-other ordering that would come from the ModuleManager approach.

We can then have a way to chain these micro-functions so that they run one after the other. Modules may define multiple ante micro-functions and then also provide a default per-module AnteHandler that implements a default, suggested order for these micro-functions.

Users can order the AnteHandlers easily by simply using the ModuleManager. The ModuleManager will take in a list of AnteHandlers and return a single AnteHandler that runs each AnteHandler in the order of the list provided. If the user is comfortable with the default ordering of each module, this is as simple as providing a list with each module's antehandler (exactly the same as BeginBlocker and EndBlocker).

If however, users wish to change the order or add, modify, or delete ante micro-functions in anyway; they can always define their own ante micro-functions and add them explicitly to the list that gets passed into module manager.

## **Default Workflow**

This is an example of a user's AnteHandler if they choose not to make any custom micro-functions.

## **Cosmos SDK code**

```

// Chains together a list of AnteHandler micro-functions that get run one after the
// other.
// Returned AnteHandler will abort on first error.
func Chainer(order []AnteHandler) AnteHandler {
 return func(ctx Context, tx Tx, simulate bool) (newCtx Context, err error) {

```

```

 for _, ante := range order {
 ctx, err := ante(ctx, tx, simulate)
 if err != nil {
 return ctx, err
 }
 }
 return ctx, err
 }
}

```

```

// AnteHandler micro-function to verify signatures
func VerifySignatures(ctx Context, tx Tx, simulate bool) (newCtx Context, err error) {
 // verify signatures
 // Returns InvalidSignature Result and abort=true if sigs invalid
 // Return OK result and abort=false if sigs are valid
}

// AnteHandler micro-function to validate memo
func ValidateMemo(ctx Context, tx Tx, simulate bool) (newCtx Context, err error) {
 // validate memo
}

// Auth defines its own default ante-handler by chaining its micro-functions in a
// recommended order
AuthModuleAnteHandler := Chainer([]AnteHandler{VerifySignatures, ValidateMemo})

```

```

// Distribution micro-function to deduct fees from tx
func DeductFees(ctx Context, tx Tx, simulate bool) (newCtx Context, err error) {
 // Deduct fees from tx
 // Abort if insufficient funds in account to pay for fees
}

// Distribution micro-function to check if fees > mempool parameter
func CheckMempoolFees(ctx Context, tx Tx, simulate bool) (newCtx Context, err error) {
 // If CheckTx: Abort if the fees are less than the mempool's minFee parameter
}

// Distribution defines its own default ante-handler by chaining its micro-functions
// in a recommended order
DistrModuleAnteHandler := Chainer([]AnteHandler{CheckMempoolFees, DeductFees})

```

```

type ModuleManager struct {
 // other fields
 AnteHandlerOrder []AnteHandler
}

func (mm ModuleManager) GetAnteHandler() AnteHandler {

```

```
 return Chainer(mm.AnteHandlerOrder)
}
```

## User Code

```
// Note: Since user is not making any custom modifications, we can just
SetAnteHandlerOrder with the default AnteHandlers provided by each module in our
preferred order
moduleManager.SetAnteHandlerOrder([]AnteHandler(AuthModuleAnteHandler,
DistrModuleAnteHandler))

app.SetAnteHandler(mm.GetAnteHandler())
```

## Custom Workflow

This is an example workflow for a user that wants to implement custom antehandler logic. In this example, the user wants to implement custom signature verification and change the order of antehandler so that validate memo runs before signature verification.

## User Code

```
// User can implement their own custom signature verification antehandler micro-
function
func CustomSigVerify(ctx Context, tx Tx, simulate bool) (newCtx Context, err error)
{
 // do some custom signature verification logic
}
```

```
// Micro-functions allow users to change order of when they get executed, and swap
out default ante-functionality with their own custom logic.
// Note that users can still chain the default distribution module handler, and auth
micro-function along with their custom ante function
moduleManager.SetAnteHandlerOrder([]AnteHandler(ValidateMemo, CustomSigVerify,
DistrModuleAnteHandler))
```

Pros:

1. Allows for ante functionality to be as modular as possible.
2. For users that do not need custom ante-functionality, there is little difference between how antehandlers work and how BeginBlock and EndBlock work in ModuleManager.
3. Still easy to understand

Cons:

1. Cannot wrap antehandlers with decorators like you can with Weave.

## Simple Decorators

This approach takes inspiration from Weave's decorator design while trying to minimize the number of breaking changes to the Cosmos SDK and maximizing simplicity. Like Weave decorators, this approach allows one `AnteDecorator` to wrap the next `AnteHandler` to do pre- and post-processing on the result.

This is useful since decorators can do defer/cleanups after an `AnteHandler` returns as well as perform some

setup beforehand. Unlike Weave decorators, these `AnteDecorator` functions can only wrap over the `AnteHandler` rather than the entire handler execution path. This is deliberate as we want decorators from different modules to perform authentication/validation on a `tx`. However, we do not want decorators being capable of wrapping and modifying the results of a `MsgHandler`.

In addition, this approach will not break any core Cosmos SDK API's. Since we preserve the notion of an `AnteHandler` and still set a single `AnteHandler` in `baseapp`, the decorator is simply an additional approach available for users that desire more customization. The API of modules (namely `x/auth`) may break with this approach, but the core API remains untouched.

Allow Decorator interface that can be chained together to create a Cosmos SDK `AnteHandler`.

This allows users to choose between implementing an `AnteHandler` by themselves and setting it in the `baseapp`, or use the decorator pattern to chain their custom decorators with the Cosmos SDK provided decorators in the order they wish.

```
// An AnteDecorator wraps an AnteHandler, and can do pre- and post-processing on the
// next AnteHandler
type AnteDecorator interface {
 AnteHandle(ctx Context, tx Tx, simulate bool, next AnteHandler) (newCtx Context,
 err error)
}
```

```
// ChainAnteDecorators will recursively link all of the AnteDecorators in the chain
// and return a final AnteHandler function
// This is done to preserve the ability to set a single AnteHandler function in the
// baseapp.
func ChainAnteDecorators(chain ...AnteDecorator) AnteHandler {
 if len(chain) == 1 {
 return func(ctx Context, tx Tx, simulate bool) {
 chain[0].AnteHandle(ctx, tx, simulate, nil)
 }
 }
 return func(ctx Context, tx Tx, simulate bool) {
 chain[0].AnteHandle(ctx, tx, simulate, ChainAnteDecorators(chain[1:]))
 }
}
```

## Example Code

Define `AnteDecorator` functions

```
// Setup GasMeter, catch OutOfGasPanic and handle appropriately
type SetUpContextDecorator struct{}

func (sud SetUpContextDecorator) AnteHandle(ctx Context, tx Tx, simulate bool, next
AnteHandler) (newCtx Context, err error) {
 ctx.GasMeter = NewGasMeter(tx.Gas)

 defer func() {
 // recover from OutOfGas panic and handle appropriately
 }
```

```

 }

 return next(ctx, tx, simulate)
}

// Signature Verification decorator. Verify Signatures and move on
type SigVerifyDecorator struct{

func (svd SigVerifyDecorator) AnteHandle(ctx Context, tx Tx, simulate bool, next
AnteHandler) (newCtx Context, err error) {
 // verify sigs. Return error if invalid

 // call next antehandler if sigs ok
 return next(ctx, tx, simulate)
}

// User-defined Decorator. Can choose to pre- and post-process on AnteHandler
type UserDefinedDecorator struct{
 // custom fields
}

func (udd UserDefinedDecorator) AnteHandle(ctx Context, tx Tx, simulate bool, next
AnteHandler) (newCtx Context, err error) {
 // pre-processing logic

 ctx, err = next(ctx, tx, simulate)

 // post-processing logic
}

```

Link AnteDecorators to create a final AnteHandler. Set this AnteHandler in baseapp.

```

// Create final antehandler by chaining the decorators together
antehandler := ChainAnteDecorators(NewSetUpContextDecorator(),
NewSigVerifyDecorator(), NewUserDefinedDecorator())

// Set chained Antehandler in the baseapp
bapp.SetAnteHandler(antehandler)

```

Pros:

1. Allows one decorator to pre- and post-process the next AnteHandler, similar to the Weave design.
2. Do not need to break baseapp API. Users can still set a single AnteHandler if they choose.

Cons:

1. Decorator pattern may have a deeply nested structure that is hard to understand, this is mitigated by having the decorator order explicitly listed in the `ChainAnteDecorators` function.
2. Does not make use of the ModuleManager design. Since this is already being used for `BeginBlocker/EndBlocker`, this proposal seems unaligned with that design pattern.

## Consequences

Since pros and cons are written for each approach, it is omitted from this section

## References

- [#4572](#): Modular AnteHandler Issue
- [#4582](#): Initial Implementation of Per-Module AnteHandler Approach
- [Weave Decorator Code](#)
- [Weave Design Videos](#)

# ADR 011: Generalize Genesis Accounts

## Changelog

- 2019-08-30: initial draft

## Context

Currently, the Cosmos SDK allows for custom account types; the `auth` keeper stores any type fulfilling its `Account` interface. However `auth` does not handle exporting or loading accounts to/from a genesis file, this is done by `genaccounts`, which only handles one of 4 concrete account types (`BaseAccount`, `ContinuousVestingAccount`, `DelayedVestingAccount` and `ModuleAccount`).

Projects desiring to use custom accounts (say custom vesting accounts) need to fork and modify `genaccounts`.

## Decision

In summary, we will (un)marshal all accounts (interface types) directly using amino, rather than converting to `genaccounts`'s `GenesisAccount` type. Since doing this removes the majority of `genaccounts`'s code, we will merge `genaccounts` into `auth`. Marshalled accounts will be stored in `auth`'s genesis state.

Detailed changes:

### 1) (Un)Marshal accounts directly using amino

The `auth` module's `GenesisState` gains a new field `Accounts`. Note these aren't of type `exported.Account` for reasons outlined in section 3.

```
// GenesisState - all auth state that must be provided at genesis
type GenesisState struct {
 Params Params `json:"params" yaml:"params"`
 Accounts []GenesisAccount `json:"accounts" yaml:"accounts"`
}
```

Now `auth`'s `InitGenesis` and `ExportGenesis` (un)marshal accounts as well as the defined params.

```

// InitGenesis - Init store state from genesis data
func InitGenesis(ctx sdk.Context, ak AccountKeeper, data GenesisState) {
 ak.SetParams(ctx, data.Params)
 // load the accounts
 for _, a := range data.Accounts {
 acc := ak.NewAccount(ctx, a) // set account number
 ak.SetAccount(ctx, acc)
 }
}

// ExportGenesis returns a GenesisState for a given context and keeper
func ExportGenesis(ctx sdk.Context, ak AccountKeeper) GenesisState {
 params := ak.GetParams(ctx)

 var genAccounts []exported.GenesisAccount
 ak.IterateAccounts(ctx, func(account exported.Account) bool {
 genAccount := account.(exported.GenesisAccount)
 genAccounts = append(genAccounts, genAccount)
 return false
 })

 return NewGenesisState(params, genAccounts)
}

```

## 2) Register custom account types on the `auth` codec

The `auth` codec must have all custom account types registered to marshal them. We will follow the pattern established in `gov` for proposals.

An example custom account definition:

```

import authtypes "github.com/cosmos/cosmos-sdk/x/auth/types"

// Register the module account type with the auth module codec so it can decode
module accounts stored in a genesis file
func init() {
 authtypes.RegisterAccountTypeCodec(ModuleAccount{}, "cosmos-sdk/ModuleAccount")
}

type ModuleAccount struct {
 ...
}

```

The `auth` codec definition:

```

var ModuleCdc *codec.LegacyAmino

func init() {
 ModuleCdc = codec.NewLegacyAmino()
 // register module msg's and Account interface
 ...
}

```

```

 // leave the codec unsealed
}

// RegisterAccountTypeCodec registers an external account type defined in another
// module for the internal ModuleCdc.
func RegisterAccountTypeCodec(o interface{}, name string) {
 ModuleCdc.RegisterConcrete(o, name, nil)
}

```

### 3) Genesis validation for custom account types

Modules implement a `ValidateGenesis` method. As `auth` does not know of account implementations, accounts will need to validate themselves.

We will unmarshal accounts into a `GenesisAccount` interface that includes a `Validate` method.

```

type GenesisAccount interface {
 exported.Account
 Validate() error
}

```

Then the `auth` `ValidateGenesis` function becomes:

```

// ValidateGenesis performs basic validation of auth genesis data returning an
// error for any failed validation criteria.
func ValidateGenesis(data GenesisState) error {
 // Validate params
 ...

 // Validate accounts
 addrMap := make(map[string]bool, len(data.Accounts))
 for _, acc := range data.Accounts {

 // check for duplicated accounts
 addrStr := acc.GetAddress().String()
 if _, ok := addrMap[addrStr]; ok {
 return fmt.Errorf("duplicate account found in genesis state; address: %s", addrStr)
 }
 addrMap[addrStr] = true

 // check account specific validation
 if err := acc.Validate(); err != nil {
 return fmt.Errorf("invalid account found in genesis state; address: %s, error: %s", addrStr, err.Error())
 }
 }
 return nil
}

```

## 4) Move add-genesis-account cli to auth

The `genaccounts` module contains a cli command to add base or vesting accounts to a genesis file.

This will be moved to `auth`. We will leave it to projects to write their own commands to add custom accounts. An extensible cli handler, similar to `gov`, could be created but it is not worth the complexity for this minor use case.

## 5) Update module and vesting accounts

Under the new scheme, module and vesting account types need some minor updates:

- Type registration on `auth`'s codec (shown above)
- A `Validate` method for each `Account` concrete type

## Status

Proposed

## Consequences

### Positive

- custom accounts can be used without needing to fork `genaccounts`
- reduction in lines of code

### Negative

### Neutral

- `genaccounts` module no longer exists
- accounts in genesis files are stored under `accounts` in `auth` rather than in the `genaccounts` module. - `add-genesis-account` cli command now in `auth`

## References

# ADR 012: State Accessors

## Changelog

- 2019 Sep 04: Initial draft

## Context

Cosmos SDK modules currently use the `KVStore` interface and `Codec` to access their respective state. While this provides a large degree of freedom to module developers, it is hard to modularize and the UX is mediocre.

First, each time a module tries to access the state, it has to marshal the value and set or get the value and finally unmarshal. Usually this is done by declaring `Keeper.GetXXX` and `Keeper.SetXXX` functions, which are repetitive and hard to maintain.

Second, this makes it harder to align with the object capability theorem: the right to access the state is defined as a `StoreKey`, which gives full access on the entire Merkle tree, so a module cannot send the access right to a specific key-value pair (or a set of key-value pairs) to another module safely.

Finally, because the getter/setter functions are defined as methods of a module's `Keeper`, the reviewers have to consider the whole Merkle tree space when they reviewing a function accessing any part of the state. There is no static way to know which part of the state that the function is accessing (and which is not).

## Decision

We will define a type named `Value`:

```
type Value struct {
 m Mapping
 key []byte
}
```

The `Value` works as a reference for a key-value pair in the state, where `Value.m` defines the key-value space it will access and `Value.key` defines the exact key for the reference.

We will define a type named `Mapping`:

```
type Mapping struct {
 storeKey sdk.StoreKey
 cdc *codec.LegacyAmino
 prefix []byte
}
```

The `Mapping` works as a reference for a key-value space in the state, where `Mapping.storeKey` defines the IAVL (sub-)tree and `Mapping.prefix` defines the optional subspace prefix.

We will define the following core methods for the `Value` type:

```
// Get and unmarshal stored data, noop if not exists, panic if cannot unmarshal
func (Value) Get(ctx Context, ptr interface{}) {}

// Get and unmarshal stored data, return error if not exists or cannot unmarshal
func (Value) GetSafe(ctx Context, ptr interface{}) {}

// Get stored data as raw byte slice
func (Value) GetRaw(ctx Context) []byte {}

// Marshal and set a raw value
func (Value) Set(ctx Context, o interface{}) {}

// Check if a raw value exists
func (Value) Exists(ctx Context) bool {}
```

```
// Delete a raw value value
func (Value) Delete(ctx Context) {}
```

We will define the following core methods for the `Mapping` type:

```
// Constructs key-value pair reference corresponding to the key argument in the
Mapping space
func (Mapping) Value(key []byte) Value {}

// Get and unmarshal stored data, noop if not exists, panic if cannot unmarshal
func (Mapping) Get(ctx Context, key []byte, ptr interface{}) {}

// Get and unmarshal stored data, return error if not exists or cannot unmarshal
func (Mapping) GetSafe(ctx Context, key []byte, ptr interface{}) {}

// Get stored data as raw byte slice
func (Mapping) GetRaw(ctx Context, key []byte) []byte {}

// Marshal and set a raw value
func (Mapping) Set(ctx Context, key []byte, o interface{}) {}

// Check if a raw value exists
func (Mapping) Has(ctx Context, key []byte) bool {}

// Delete a raw value value
func (Mapping) Delete(ctx Context, key []byte) {}
```

Each method of the `Mapping` type that is passed the arguments `ctx`, `key`, and `args...` will proxy the call to `Mapping.Value(key)` with arguments `ctx` and `args...`.

In addition, we will define and provide a common set of types derived from the `Value` type:

```
type Boolean struct { Value }
type Enum struct { Value }
type Integer struct { Value; enc IntEncoding }
type String struct { Value }
// ...
```

Where the encoding schemes can be different, `o` arguments in core methods are typed, and `ptr` arguments in core methods are replaced by explicit return types.

Finally, we will define a family of types derived from the `Mapping` type:

```
type Indexer struct {
 m Mapping
 enc IntEncoding
}
```

Where the `key` argument in core method is typed.

Some of the properties of the accessor types are:

- State access happens only when a function which takes a `Context` as an argument is invoked
- Accessor type structs give rights to access the state only that the struct is referring, no other
- Marshalling/Unmarshalling happens implicitly within the core methods

## Status

Proposed

## Consequences

### Positive

- Serialization will be done automatically
- Shorter code size, less boilerplate, better UX
- References to the state can be transferred safely
- Explicit scope of accessing

### Negative

- Serialization format will be hidden
- Different architecture from the current, but the use of accessor types can be opt-in
- Type-specific types (e.g. `Boolean` and `Integer`) have to be defined manually

### Neutral

## References

- [#4554](#)

# ADR 013: Observability

## Changelog

- 20-01-2020: Initial Draft

## Status

Proposed

## Context

Telemetry is paramount into debugging and understanding what the application is doing and how it is performing. We aim to expose metrics from modules and other core parts of the Cosmos SDK.

In addition, we should aim to support multiple configurable sinks that an operator may choose from. By default, when telemetry is enabled, the application should track and expose metrics that are stored in-memory. The operator may choose to enable additional sinks, where we support only [Prometheus](#) for now, as it's battle-tested, simple to setup, open source, and is rich with ecosystem tooling.

We must also aim to integrate metrics into the Cosmos SDK in the most seamless way possible such that metrics may be added or removed at will and without much friction. To do this, we will use the [go-metrics](#) library.

Finally, operators may enable telemetry along with specific configuration options. If enabled, metrics will be exposed via `/metrics?format={text|prometheus}` via the API server.

## Decision

We will add an additional configuration block to `app.toml` that defines telemetry settings:

```
#####
Telemetry Configuration
#####

[telemetry]

Prefixed with keys to separate services
service-name = {{ .Telemetry.ServiceName }}

Enabled enables the application telemetry functionality. When enabled,
an in-memory sink is also enabled by default. Operators may also enable
other sinks such as Prometheus.
enabled = {{ .Telemetry.Enabled }}

Enable prefixing gauge values with hostname
enable-hostname = {{ .Telemetry.EnableHostname }}

Enable adding hostname to labels
enable-hostname-label = {{ .Telemetry.EnableHostnameLabel }}

Enable adding service to labels
enable-service-label = {{ .Telemetry.EnableServiceLabel }}

PrometheusRetentionTime, when positive, enables a Prometheus metrics sink.
prometheus-retention-time = {{ .Telemetry.PrometheusRetentionTime }}
```

The given configuration allows for two sinks -- in-memory and Prometheus. We create a `Metrics` type that performs all the bootstrapping for the operator, so capturing metrics becomes seamless.

```
// Metrics defines a wrapper around application telemetry functionality. It allows
// metrics to be gathered at any point in time. When creating a Metrics object,
// internally, a global metrics is registered with a set of sinks as configured
// by the operator. In addition to the sinks, when a process gets a SIGUSR1, a
// dump of formatted recent metrics will be sent to STDERR.
type Metrics struct {
 memSink *metrics.InmemSink
 prometheusEnabled bool
}

// Gather collects all registered metrics and returns a GatherResponse where the
// metrics are encoded depending on the type. Metrics are either encoded via
// Prometheus or JSON if in-memory.
func (m *Metrics) Gather(format string) (GatherResponse, error) {
```

```

switch format {
case FormatPrometheus:
 return m.gatherPrometheus()

case FormatText:
 return m.gatherGeneric()

case FormatDefault:
 return m.gatherGeneric()

default:
 return GatherResponse{}, fmt.Errorf("unsupported metrics format: %s", format)
}
}

```

In addition, `Metrics` allows us to gather the current set of metrics at any given point in time. An operator may also choose to send a signal, SIGUSR1, to dump and print formatted metrics to STDERR.

During an application's bootstrapping and construction phase, if `Telemetry.Enabled` is `true`, the API server will create an instance of a reference to `Metrics` object and will register a metrics handler accordingly.

```

func (s *Server) Start(cfg config.Config) error {
// ...

if cfg.Telemetry.Enabled {
 m, err := telemetry.New(cfg.Telemetry)
 if err != nil {
 return err
 }

 s.metrics = m
 s.registerMetrics()
}

// ...
}

func (s *Server) registerMetrics() {
 metricsHandler := func(w http.ResponseWriter, r *http.Request) {
 format := strings.TrimSpace(r.FormValue("format"))

 gr, err := s.metrics.Gather(format)
 if err != nil {
 rest.WriteErrorResponse(w, http.StatusBadRequest, fmt.Sprintf("failed to
gather metrics: %s", err))
 return
 }

 w.Header().Set("Content-Type", gr.ContentType)
 _, _ = w.Write(gr.Metrics)
 }
}

```

```
}

 s.Router.HandleFunc("/metrics", metricsHandler).Methods("GET")
}


```

Application developers may track counters, gauges, summaries, and key/value metrics. There is no additional lifting required by modules to leverage profiling metrics. To do so, it's as simple as:

```
func (k BaseKeeper) MintCoins(ctx sdk.Context, moduleName string, amt sdk.Coins)
error {
 defer metrics.MeasureSince(time.Now(), "MintCoins")
 // ...
}
```

## Consequences

### Positive

- Exposure into the performance and behavior of an application

### Negative

### Neutral

## References

# ADR 14: Proportional Slashing

## Changelog

- 2019-10-15: Initial draft
- 2020-05-25: Removed correlation root slashing
- 2020-07-01: Updated to include S-curve function instead of linear

## Context

In Proof of Stake-based chains, centralization of consensus power amongst a small set of validators can cause harm to the network due to increased risk of censorship, liveness failure, fork attacks, etc. However, while this centralization causes a negative externality to the network, it is not directly felt by the delegators contributing towards delegating towards already large validators. We would like a way to pass on the negative externality cost of centralization onto those large validators and their delegators.

## Decision

### Design

To solve this problem, we will implement a procedure called Proportional Slashing. The desire is that the larger a validator is, the more they should be slashed. The first naive attempt is to make a validator's slash percent proportional to their share of consensus voting power.

```
slash_amount = k * power // power is the faulting validator's voting power and k is some on-chain constant
```

However, this will incentivize validators with large amounts of stake to split up their voting power amongst accounts (sybil attack), so that if they fault, they all get slashed at a lower percent. The solution to this is to take into account not just a validator's own voting percentage, but also the voting percentage of all the other validators who get slashed in a specified time frame.

```
slash_amount = k * (power_1 + power_2 + ... + power_n) // where power_i is the voting power of the ith validator faulting in the specified time frame and k is some on-chain constant
```

Now, if someone splits a validator of 10% into two validators of 5% each which both fault, then they both fault in the same time frame, they both will get slashed at the sum 10% amount.

However in practice, we likely don't want a linear relation between amount of stake at fault, and the percentage of stake to slash. In particular, solely 5% of stake double signing effectively did nothing to majorly threaten security, whereas 30% of stake being at fault clearly merits a large slashing factor, due to being very close to the point at which Tendermint security is threatened. A linear relation would require a factor of 6 gap between these two, whereas the difference in risk posed to the network is much larger. We propose using S-curves (formally [logistic functions](#) to solve this). S-Curves capture the desired criterion quite well. They allow the slashing factor to be minimal for small values, and then grow very rapidly near some threshold point where the risk posed becomes notable.

### Parameterization

This requires parameterizing a logistic function. It is very well understood how to parameterize this. It has four parameters:

1. A minimum slashing factor
2. A maximum slashing factor
3. The inflection point of the S-curve (essentially where do you want to center the S)
4. The rate of growth of the S-curve (How elongated is the S)

### Correlation across non-sybil validators

One will note, that this model doesn't differentiate between multiple validators run by the same operators vs validators run by different operators. This can be seen as an additional benefit in fact. It incentivizes validators to differentiate their setups from other validators, to avoid having correlated faults with them or else they risk a higher slash. So for example, operators should avoid using the same popular cloud hosting platforms or using the same Staking as a Service providers. This will lead to a more resilient and decentralized network.

### Griefing

Griefing, the act of intentionally getting oneself slashed in order to make another's slash worse, could be a concern here. However, using the protocol described here, the attacker also gets equally impacted by the grief as the victim, so it would not provide much benefit to the griefer.

### Implementation

In the slashing module, we will add two queues that will track all of the recent slash events. For double sign faults, we will define "recent slashes" as ones that have occurred within the last unbonding period . For

liveness faults, we will define "recent slashes" as ones that have occurred within the last `jail` period .

```
type SlashEvent struct {
 Address sdk.ValAddress
 ValidatorVotingPercent sdk.Dec
 SlashedSoFar sdk.Dec
}
```

These slash events will be pruned from the queue once they are older than their respective "recent slash period".

Whenever a new slash occurs, a `SlashEvent` struct is created with the faulting validator's voting percent and a `SlashedSoFar` of 0. Because recent slash events are pruned before the unbonding period andunjail period expires, it should not be possible for the same validator to have multiple `SlashEvents` in the same Queue at the same time.

We then will iterate over all the `SlashEvents` in the queue, adding their `ValidatorVotingPercent` to calculate the new percent to slash all the validators in the queue at, using the "Square of Sum of Roots" formula introduced above.

Once we have the `NewSlashPercent`, we then iterate over all the `SlashEvent`s in the queue once again, and if `NewSlashPercent > SlashedSoFar` for that `SlashEvent`, we call the `staking.Slash(slashEvent.Address, slashEvent.Power, Math.Min(Math.Max(minSlashPercent, NewSlashPercent - SlashedSoFar), maxSlashPercent))` (we pass in the power of the validator before any slashes occurred, so that we slash the right amount of tokens). We then set `SlashEvent.SlashedSoFar` amount to `NewSlashPercent`.

## Status

Proposed

## Consequences

### Positive

- Increases decentralization by disincentivizing delegating to large validators
- Incentivizes Decorrelation of Validators
- More severely punishes attacks than accidental faults
- More flexibility in slashing rates parameterization

### Negative

- More computationally expensive than current implementation. Will require more data about "recent slashing events" to be stored on chain.

## ADR 016: Validator Consensus Key Rotation

### Changelog

- 2019 Oct 23: Initial draft
- 2019 Nov 28: Add key rotation fee

## Context

Validator consensus key rotation feature has been discussed and requested for a long time, for the sake of safer validator key management policy (e.g. <https://github.com/tendermint/tendermint/issues/1136>). So, we suggest one of the simplest form of validator consensus key rotation implementation mostly onto Cosmos SDK.

We don't need to make any update on consensus logic in Tendermint because Tendermint does not have any mapping information of consensus key and validator operator key, meaning that from Tendermint point of view, a consensus key rotation of a validator is simply a replacement of a consensus key to another.

Also, it should be noted that this ADR includes only the simplest form of consensus key rotation without considering multiple consensus keys concept. Such multiple consensus keys concept shall remain a long term goal of Tendermint and Cosmos SDK.

## Decision

### Pseudo procedure for consensus key rotation

- create new random consensus key.
- create and broadcast a transaction with a `MsgRotateConsPubKey` that states the new consensus key is now coupled with the validator operator with signature from the validator's operator key.
- old consensus key becomes unable to participate on consensus immediately after the update of key mapping state on-chain.
- start validating with new consensus key.
- validators using HSM and KMS should update the consensus key in HSM to use the new rotated key after the height `h` when `MsgRotateConsPubKey` committed to the blockchain.

### Considerations

- consensus key mapping information management strategy
  - store history of each key mapping changes in the kvstore.
  - the state machine can search corresponding consensus key paired with given validator operator for any arbitrary height in a recent unbonding period.
  - the state machine does not need any historical mapping information which is past more than unbonding period.
- key rotation costs related to LCD and IBC
  - LCD and IBC will have traffic/computation burden when there exists frequent power changes
  - In current Tendermint design, consensus key rotations are seen as power changes from LCD or IBC perspective
  - Therefore, to minimize unnecessary frequent key rotation behavior, we limited maximum number of rotation in recent unbonding period and also applied exponentially increasing rotation fee
- limits
  - a validator cannot rotate its consensus key more than `MaxConsPubKeyRotations` time for any unbonding period, to prevent spam.
  - parameters can be decided by governance and stored in genesis file.
- key rotation fee
  - a validator should pay `KeyRotationFee` to rotate the consensus key which is calculated as below

- o `KeyRotationFee = (max( VotingPowerPercentage 100, 1) InitialKeyRotationFee ) * 2^(number of rotations in ConsPubKeyRotationHistory in recent unbonding period)`
- evidence module
  - o evidence module can search corresponding consensus key for any height from slashing keeper so that it can decide which consensus key is supposed to be used for given height.
- abci.ValidatorUpdate
  - o tendermint already has ability to change a consensus key by ABCI communication(`ValidatorUpdate`).
  - o validator consensus key update can be done via creating new + delete old by change the power to zero.
  - o therefore, we expect we even do not need to change tendermint codebase at all to implement this feature.
- new genesis parameters in `staking` module
  - o `MaxConsPubKeyRotations` : maximum number of rotation can be executed by a validator in recent unbonding period. default value 10 is suggested(11th key rotation will be rejected)
  - o `InitialKeyRotationFee` : the initial key rotation fee when no key rotation has happened in recent unbonding period. default value 1atom is suggested(1atom fee for the first key rotation in recent unbonding period)

## Workflow

1. The validator generates a new consensus keypair.
2. The validator generates and signs a `MsgRotateConsPubKey` tx with their operator key and new `ConsPubKey`

```
type MsgRotateConsPubKey struct {
 ValidatorAddress sdk.ValAddress
 NewPubKey crypto.PubKey
}
```

3. `handleMsgRotateConsPubKey` gets `MsgRotateConsPubKey`, calls `RotateConsPubKey` with emits event
4. `RotateConsPubKey`

- o checks if `NewPubKey` is not duplicated on `ValidatorsByConsAddr`
- o checks if the validator is does not exceed parameter `MaxConsPubKeyRotations` by iterating `ConsPubKeyRotationHistory`
- o checks if the signing account has enough balance to pay `KeyRotationFee`
- o pays `KeyRotationFee` to community fund
- o overwrites `NewPubKey` in `validator.ConsPubKey`
- o deletes old `ValidatorByConsAddr`
- o `SetValidatorByConsAddr` for `NewPubKey`
- o Add `ConsPubKeyRotationHistory` for tracking rotation

```
type ConsPubKeyRotationHistory struct {
 OperatorAddress sdk.ValAddress
 OldConsPubKey crypto.PubKey
```

```

 NewConsPubKey crypto.PubKey
 RotatedHeight int64
}

```

5. `ApplyAndReturnValidatorSetUpdates` checks if there is `ConsPubKeyRotationHistory` with `ConsPubKeyRotationHistory.RotatedHeight == ctx.BlockHeight()` and if so, generates 2 `ValidatorUpdate`, one for a remove validator and one for create new validator

```

abci.ValidatorUpdate{
 PubKey: cmtytypes.TM2PB.PubKey(OldConsPubKey),
 Power: 0,
}

abci.ValidatorUpdate{
 PubKey: cmtytypes.TM2PB.PubKey(NewConsPubKey),
 Power: v.ConsensusPower(),
}

```

6. at `previousVotes` Iteration logic of `AllocateTokens`, `previousVote` using `OldConsPubKey` match up with `ConsPubKeyRotationHistory`, and replace validator for token allocation

7. Migrate `ValidatorSigningInfo` and `ValidatorMissedBlockBitArray` from `OldConsPubKey` to `NewConsPubKey`

- Note : All above features shall be implemented in `staking` module.

## Status

Proposed

## Consequences

### Positive

- Validators can immediately or periodically rotate their consensus key to have better security policy
- improved security against Long-Range attacks (<https://nearprotocol.com/blog/long-range-attacks-and-a-new-fork-choice-rule>) given a validator throws away the old consensus key(s)

### Negative

- Slash module needs more computation because it needs to lookup corresponding consensus key of validators for each height
- frequent key rotations will make light client bisection less efficient

### Neutral

## References

- on tendermint repo : <https://github.com/tendermint/tendermint/issues/1136>
- on cosmos-sdk repo : <https://github.com/cosmos/cosmos-sdk/issues/5231>
- about multiple consensus keys :  
<https://github.com/tendermint/tendermint/issues/1758#issuecomment-545291698>

# ADR 17: Historical Header Module

## Changelog

- 26 November 2019: Start of first version
- 2 December 2019: Final draft of first version

## Context

In order for the Cosmos SDK to implement the [IBC specification](#), modules within the Cosmos SDK must have the ability to introspect recent consensus states (validator sets & commitment roots) as proofs of these values on other chains must be checked during the handshakes.

## Decision

The application MUST store the most recent `n` headers in a persistent store. At first, this store MAY be the current Merklised store. A non-Merklised store MAY be used later as no proofs are necessary.

The application MUST store this information by storing new headers immediately when handling `abci.RequestBeginBlock`:

```
func BeginBlock(ctx sdk.Context, keeper HistoricalHeaderKeeper, req
abci.RequestBeginBlock) abci.ResponseBeginBlock {
 info := HistoricalInfo{
 Header: ctx.BlockHeader(),
 ValSet: keeper.StakingKeeper.GetAllValidators(ctx), // note that this must be
 stored in a canonical order
 }
 keeper.SetHistoricalInfo(ctx, ctx.BlockHeight(), info)
 n := keeper.GetParamRecentHeadersToStore()
 keeper.PruneHistoricalInfo(ctx, ctx.BlockHeight() - n)
 // continue handling request
}
```

Alternatively, the application MAY store only the hash of the validator set.

The application MUST make these past `n` committed headers available for querying by Cosmos SDK modules through the `Keeper`'s `GetHistoricalInfo` function. This MAY be implemented in a new module, or it MAY also be integrated into an existing one (likely `x/staking` or `x/ibc`).

`n` MAY be configured as a parameter store parameter, in which case it could be changed by `ParameterChangeProposal`s, although it will take some blocks for the stored information to catch up if `n` is increased.

## Status

Proposed.

## Consequences

Implementation of this ADR will require changes to the Cosmos SDK. It will not require changes to Tendermint.

## Positive

- Easy retrieval of headers & state roots for recent past heights by modules anywhere in the Cosmos SDK.
- No RPC calls to Tendermint required.
- No ABCI alterations required.

## Negative

- Duplicates `n` headers data in Tendermint & the application (additional disk usage) - in the long term, an approach such as [this](#) might be preferable.

## Neutral

(none known)

## References

- [ICS 2: "Consensus state introspection"](#)

# ADR 18: Extendable Voting Periods

## Changelog

- 1 January 2020: Start of first version

## Context

Currently the voting period for all governance proposals is the same. However, this is suboptimal as all governance proposals do not require the same time period. For more non-contentious proposals, they can be dealt with more efficiently with a faster period, while more contentious or complex proposals may need a longer period for extended discussion/consideration.

## Decision

We would like to design a mechanism for making the voting period of a governance proposal variable based on the demand of voters. We would like it to be based on the view of the governance participants, rather than just the proposer of a governance proposal (thus, allowing the proposer to select the voting period length is not sufficient).

However, we would like to avoid the creation of an entire second voting process to determine the length of the voting period, as it just pushed the problem to determining the length of that first voting period.

Thus, we propose the following mechanism:

## Params

- The current gov param `VotingPeriod` is to be replaced by a `MinVotingPeriod` param. This is the default voting period that all governance proposal voting periods start with.
- There is a new gov param called `MaxVotingPeriodExtension`.

## Mechanism

There is a new `Msg` type called `MsgExtendVotingPeriod`, which can be sent by any staked account during a proposal's voting period. It allows the sender to unilaterally extend the length of the voting period by `MaxVotingPeriodExtension * sender's share of voting power`. Every address can only call `MsgExtendVotingPeriod` once per proposal.

So for example, if the `MaxVotingPeriodExtension` is set to 100 Days, then anyone with 1% of voting power can extend the voting power by 1 day. If 33% of voting power has sent the message, the voting period will be extended by 33 days. Thus, if absolutely everyone chooses to extend the voting period, the absolute maximum voting period will be `MinVotingPeriod + MaxVotingPeriodExtension`.

This system acts as a sort of distributed coordination, where individual stakers choosing to extend or not, allows the system to gauge the contentiousness/complexity of the proposal. It is extremely unlikely that many stakers will choose to extend at the exact same time, it allows stakers to view how long others have already extended thus far, to decide whether or not to extend further.

## Dealing with Unbonding/Redelegation

There is one thing that needs to be addressed. How to deal with redelegation/unbonding during the voting period. If a staker of 5% calls `MsgExtendVotingPeriod` and then unbonds, does the voting period then decrease by 5 days again? This is not good as it can give people a false sense of how long they have to make their decision. For this reason, we want to design it such that the voting period length can only be extended, not shortened. To do this, the current extension amount is based on the highest percent that voted extension at any time. This is best explained by example:

1. Let's say 2 stakers of voting power 4% and 3% respectively vote to extend. The voting period will be extended by 7 days.
2. Now the staker of 3% decides to unbond before the end of the voting period. The voting period extension remains 7 days.
3. Now, let's say another staker of 2% voting power decides to extend voting period. There is now 6% of active voting power choosing the extend. The voting power remains 7 days.
4. If a fourth staker of 10% chooses to extend now, there is a total of 16% of active voting power wishing to extend. The voting period will be extended to 16 days.

## Delegators

Just like votes in the actual voting period, delegators automatically inherit the extension of their validators. If their validator chooses to extend, their voting power will be used in the validator's extension. However, the delegator is unable to override their validator and "unextend" as that would contradict the "voting power length can only be ratcheted up" principle described in the previous section. However, a delegator may choose the extend using their personal voting power, if their validator has not done so.

## Status

Proposed

## Consequences

### Positive

- More complex/contentious governance proposals will have more time to properly digest and deliberate

## Negative

- Governance process becomes more complex and requires more understanding to interact with effectively
- Can no longer predict when a governance proposal will end. Can't assume order in which governance proposals will end.

## Neutral

- The minimum voting period can be made shorter

## References

- [Cosmos Forum post where idea first originated](#)

# ADR 019: Protocol Buffer State Encoding

## Changelog

- 2020 Feb 15: Initial Draft
- 2020 Feb 24: Updates to handle messages with interface fields
- 2020 Apr 27: Convert usages of `oneof` for interfaces to `Any`
- 2020 May 15: Describe `cosmos_proto` extensions and amino compatibility
- 2020 Dec 4: Move and rename `MarshalAny` and `UnmarshalAny` into the `codec.Codec` interface.
- 2021 Feb 24: Remove mentions of `HybridCodec`, which has been abandoned in [#6843](#).

## Status

Accepted

## Context

Currently, the Cosmos SDK utilizes [go-amino](#) for binary and JSON object encoding over the wire bringing parity between logical objects and persistence objects.

From the Amino docs:

*Amino is an object encoding specification. It is a subset of Proto3 with an extension for interface support. See the [Proto3 spec](#) for more information on Proto3, which Amino is largely compatible with (but not with Proto2).*

*The goal of the Amino encoding protocol is to bring parity into logic objects and persistence objects.*

Amino also aims to have the following goals (not a complete list):

- Binary bytes must be decode-able with a schema.
- Schema must be upgradeable.
- The encoder and decoder logic must be reasonably simple.

However, we believe that Amino does not fulfill these goals completely and does not fully meet the needs of a truly flexible cross-language and multi-client compatible encoding protocol in the Cosmos SDK. Namely, Amino has proven to be a big pain-point in regards to supporting object serialization across clients written in various languages while providing virtually little in the way of true backwards compatibility and

upgradeability. Furthermore, through profiling and various benchmarks, Amino has been shown to be an extremely large performance bottleneck in the Cosmos SDK<sup>1</sup>. This is largely reflected in the performance of simulations and application transaction throughput.

Thus, we need to adopt an encoding protocol that meets the following criteria for state serialization:

- Language agnostic
- Platform agnostic
- Rich client support and thriving ecosystem
- High performance
- Minimal encoded message size
- Codegen-based over reflection-based
- Supports backward and forward compatibility

Note, migrating away from Amino should be viewed as a two-pronged approach, state and client encoding. This ADR focuses on state serialization in the Cosmos SDK state machine. A corresponding ADR will be made to address client-side encoding.

## Decision

We will adopt [Protocol Buffers](#) for serializing persisted structured data in the Cosmos SDK while providing a clean mechanism and developer UX for applications wishing to continue to use Amino. We will provide this mechanism by updating modules to accept a codec interface, `Marshaler`, instead of a concrete Amino codec. Furthermore, the Cosmos SDK will provide two concrete implementations of the `Marshaler` interface: `AminoCodec` and `ProtoCodec`.

- `AminoCodec` : Uses Amino for both binary and JSON encoding.
- `ProtoCodec` : Uses Protobuf for both binary and JSON encoding.

Modules will use whichever codec that is instantiated in the app. By default, the Cosmos SDK's `simapp` instantiates a `ProtoCodec` as the concrete implementation of `Marshaler`, inside the `MakeTestEncodingConfig` function. This can be easily overwritten by app developers if they so desire.

The ultimate goal will be to replace Amino JSON encoding with Protobuf encoding and thus have modules accept and/or extend `ProtoCodec`. Until then, Amino JSON is still provided for legacy use-cases. A handful of places in the Cosmos SDK still have Amino JSON hardcoded, such as the Legacy API REST endpoints and the `x/params` store. They are planned to be converted to Protobuf in a gradual manner.

## Module Codecs

Modules that do not require the ability to work with and serialize interfaces, the path to Protobuf migration is pretty straightforward. These modules are to simply migrate any existing types that are encoded and persisted via their concrete Amino codec to Protobuf and have their keeper accept a `Marshaler` that will be a `ProtoCodec`. This migration is simple as things will just work as-is.

Note, any business logic that needs to encode primitive types like `bool` or `int64` should use [gogoproto](#) Value types.

Example:

```
ts, err := gogotypes.TimestampProto(completionTime)
if err != nil {
 // ...
}
```

```
}

bz := cdc.MustMarshal(ts)
```

However, modules can vary greatly in purpose and design and so we must support the ability for modules to be able to encode and work with interfaces (e.g. `Account` or `Content`). For these modules, they must define their own codec interface that extends `Marshaler`. These specific interfaces are unique to the module and will contain method contracts that know how to serialize the needed interfaces.

Example:

```
// x/auth/types/codec.go

type Codec interface {
 codec.Codec

 MarshalAccount(acc exported.Account) ([]byte, error)
 UnmarshalAccount(bz []byte) (exported.Account, error)

 MarshalAccountJSON(acc exported.Account) ([]byte, error)
 UnmarshalAccountJSON(bz []byte) (exported.Account, error)
}
```

## Usage of `Any` to encode interfaces

In general, module-level .proto files should define messages which encode interfaces using `google.protobuf.Any`. After [extension discussion](#), this was chosen as the preferred alternative to application-level `oneof`s as in our original protobuf design. The arguments in favor of `Any` can be summarized as follows:

- `Any` provides a simpler, more consistent client UX for dealing with interfaces than app-level `oneof`s that will need to be coordinated more carefully across applications. Creating a generic transaction signing library using `oneof`s may be cumbersome and critical logic may need to be reimplemented for each chain
- `Any` provides more resistance against human error than `oneof`
- `Any` is generally simpler to implement for both modules and apps

The main counter-argument to using `Any` centers around its additional space and possibly performance overhead. The space overhead could be dealt with using compression at the persistence layer in the future and the performance impact is likely to be small. Thus, not using `Any` is seem as a pre-mature optimization, with user experience as the higher order concern.

Note, that given the Cosmos SDK's decision to adopt the `Codec` interfaces described above, apps can still choose to use `oneof` to encode state and transactions but it is not the recommended approach. If apps do choose to use `oneof`s instead of `Any` they will likely lose compatibility with client apps that support multiple chains. Thus developers should think carefully about whether they care more about what is possibly a pre-mature optimization or end-user and client developer UX.

## Safe usage of `Any`

By default, the [gogo protobuf implementation of `Any`](#) uses [global type registration](#) to decode values packed in `Any` into concrete go types. This introduces a vulnerability where any malicious module in the dependency tree could register a type with the global protobuf registry and cause it to be loaded and unmarshaled by a transaction that referenced it in the `type_url` field.

To prevent this, we introduce a type registration mechanism for decoding `Any` values into concrete types through the `InterfaceRegistry` interface which bears some similarity to type registration with Amino:

```
type InterfaceRegistry interface {
 // RegisterInterface associates protoName as the public name for the
 // interface passed in as iface
 // Ex:
 // registry.RegisterInterface("cosmos_sdk.Msg", (*sdk.Msg)(nil))
 RegisterInterface(protoName string, iface interface{})

 // RegisterImplementations registers impls as a concrete implementations of
 // the interface iface
 // Ex:
 // registry.RegisterImplementations((*sdk.Msg)(nil), &MsgSend{},
 &MsgMultiSend{})
 RegisterImplementations(iface interface{}, impls ...proto.Message)

}
```

In addition to serving as a whitelist, `InterfaceRegistry` can also serve to communicate the list of concrete types that satisfy an interface to clients.

In .proto files:

- fields which accept interfaces should be annotated with `cosmos_proto.accepts_interface` using the same full-qualified name passed as `protoName` to  
`InterfaceRegistry.RegisterInterface`
- interface implementations should be annotated with `cosmos_proto.implements_interface` using the same full-qualified name passed as `protoName` to  
`InterfaceRegistry.RegisterInterface`

In the future, `protoName`, `cosmos_proto.accepts_interface`,  
`cosmos_proto.implements_interface` may be used via code generation, reflection &/or static linting.

The same struct that implements `InterfaceRegistry` will also implement an interface

`InterfaceUnpacker` to be used for unpacking `Any`:

```
type InterfaceUnpacker interface {
 // UnpackAny unpacks the value in any to the interface pointer passed in as
 // iface. Note that the type in any must have been registered with
 // RegisterImplementations as a concrete type for that interface
 // Ex:
 // var msg sdk.Msg
 // err := ctx.UnpackAny(any, &msg)
 // ...
}
```

```
 UnpackAny(any *Any, iface interface{}) error
}
```

Note that `InterfaceRegistry` usage does not deviate from standard protobuf usage of `Any`, it just introduces a security and introspection layer for golang usage.

`InterfaceRegistry` will be a member of `ProtoCodec` described above. In order for modules to register interface types, app modules can optionally implement the following interface:

```
type InterfaceModule interface {
 RegisterInterfaceTypes(InterfaceRegistry)
}
```

The module manager will include a method to call `RegisterInterfaceTypes` on every module that implements it in order to populate the `InterfaceRegistry`.

## Using `Any` to encode state

The Cosmos SDK will provide support methods `MarshalInterface` and `UnmarshalInterface` to hide a complexity of wrapping interface types into `Any` and allow easy serialization.

```
import "github.com/cosmos/cosmos-sdk/codec"

// note: eviexported.Evidence is an interface type
func MarshalEvidence(cdc codec.BinaryCodec, e eviexported.Evidence) ([]byte, error) {
 return cdc.MarshalInterface(e)
}

func UnmarshalEvidence(cdc codec.BinaryCodec, bz []byte) (eviexported.Evidence, error) {
 var evi eviexported.Evidence
 err := cdc.UnmarshalInterface(&evi, bz)
 return err, nil
}
```

## Using `Any` in `sdk.Msg`s

A similar concept is to be applied for messages that contain interfaces fields. For example, we can define `MsgSubmitEvidence` as follows where `Evidence` is an interface:

```
// x/evidence/types/types.proto

message MsgSubmitEvidence {
 bytes submitter = 1
 [
 (gogoproto.casttype) = "github.com/cosmos/cosmos-sdk/types.AccAddress"
];
}
```

```
 google.protobuf.Any evidence = 2;
}
```

Note that in order to unpack the evidence from `Any` we do need a reference to `InterfaceRegistry`. In order to reference evidence in methods like `ValidateBasic` which shouldn't have to know about the `InterfaceRegistry`, we introduce an `UnpackInterfaces` phase to deserialization which unpacks interfaces before they're needed.

## Unpacking Interfaces

To implement the `UnpackInterfaces` phase of deserialization which unpacks interfaces wrapped in `Any` before they're needed, we create an interface that `sdk.Msg`'s and other types can implement:

```
type UnpackInterfacesMessage interface {
 UnpackInterfaces(InterfaceUnpacker) error
}
```

We also introduce a private `cachedValue interface{}` field onto the `Any` struct itself with a public getter `GetCachedValue() interface{}`.

The `UnpackInterfaces` method is to be invoked during message deserialization right after `Unmarshal` and any interface values packed in `Any`'s will be decoded and stored in `cachedValue` for reference later.

Then unpacked interface values can safely be used in any code afterwards without knowledge of the `InterfaceRegistry` and messages can introduce a simple getter to cast the cached value to the correct interface type.

This has the added benefit that unmarshaling of `Any` values only happens once during initial deserialization rather than every time the value is read. Also, when `Any` values are first packed (for instance in a call to `NewMsgSubmitEvidence`), the original interface value is cached so that unmarshaling isn't needed to read it again.

`MsgSubmitEvidence` could implement `UnpackInterfaces`, plus a convenience getter `GetEvidence` as follows:

```
func (msg MsgSubmitEvidence) UnpackInterfaces(ctx sdk.InterfaceRegistry) error {
 var evi eviexported.Evidence
 return ctx.UnpackAny(msg.Evidence, &evi)
}

func (msg MsgSubmitEvidence) GetEvidence() eviexported.Evidence {
 return msg.Evidence.GetCachedValue().(eviexported.Evidence)
}
```

## Amino Compatibility

Our custom implementation of `Any` can be used transparently with Amino if used with the proper codec instance. What this means is that interfaces packed within `Any`'s will be amino marshaled like regular Amino interfaces (assuming they have been registered properly with Amino).

In order for this functionality to work:

- all legacy code must use `*codec.LegacyAmino` instead of `*amino.Codec` which is now a wrapper which properly handles `Any`
- all new code should use `Marshaler` which is compatible with both amino and protobuf
- Also, before v0.39, `codec.LegacyAmino` will be renamed to `codec.LegacyAmino`.

## Why Wasn't X Chosen Instead

For a more complete comparison to alternative protocols, see [here](#).

### Cap'n Proto

While [Cap'n Proto](#) does seem like an advantageous alternative to Protobuf due to its native support for interfaces/generics and built in canonicalization, it does lack the rich client ecosystem compared to Protobuf and is a bit less mature.

### FlatBuffers

[FlatBuffers](#) is also a potentially viable alternative, with the primary difference being that FlatBuffers does not need a parsing/unpacking step to a secondary representation before you can access data, often coupled with per-object memory allocation.

However, it would require great efforts into research and full understanding the scope of the migration and path forward -- which isn't immediately clear. In addition, FlatBuffers aren't designed for untrusted inputs.

## Future Improvements & Roadmap

In the future we may consider a compression layer right above the persistence layer which doesn't change tx or merkle tree hashes, but reduces the storage overhead of `Any`. In addition, we may adopt protobuf naming conventions which make type URLs a bit more concise while remaining descriptive.

Additional code generation support around the usage of `Any` is something that could also be explored in the future to make the UX for go developers more seamless.

## Consequences

### Positive

- Significant performance gains.
- Supports backward and forward type compatibility.
- Better support for cross-language clients.

### Negative

- Learning curve required to understand and implement Protobuf messages.
- Slightly larger message size due to use of `Any`, although this could be offset by a compression layer in the future

### Neutral

## References

1. <https://github.com/cosmos/cosmos-sdk/issues/4977>
2. <https://github.com/cosmos/cosmos-sdk/issues/5444>

# ADR 020: Protocol Buffer Transaction Encoding

## Changelog

- 2020 March 06: Initial Draft
- 2020 March 12: API Updates
- 2020 April 13: Added details on interface `oneof` handling
- 2020 April 30: Switch to `Any`
- 2020 May 14: Describe public key encoding
- 2020 June 08: Store `TxBody` and `AuthInfo` as bytes in `SignDoc`; Document `TxRaw` as broadcast and storage type.
- 2020 August 07: Use ADR 027 for serializing `SignDoc`.
- 2020 August 19: Move sequence field from `SignDoc` to `SignerInfo`, as discussed in [#6966](#).
- 2020 September 25: Remove `PublicKey` type in favor of `secp256k1.PubKey`,  
`ed25519.PubKey` and `multisig.LegacyAminoPubKey`.
- 2020 October 15: Add `GetAccount` and `GetAccountWithHeight` methods to the  
`AccountRetriever` interface.
- 2021 Feb 24: The Cosmos SDK does not use Tendermint's `PubKey` interface anymore, but its own  
`cryptotypes.PubKey`. Updates to reflect this.
- 2021 May 3: Rename `clientCtx.JSONMarshaler` to `clientCtx.JSONCodec`.
- 2021 June 10: Add `clientCtx.Codec: codec.Codec`.

## Status

Accepted

## Context

This ADR is a continuation of the motivation, design, and context established in [ADR 019](#), namely, we aim to design the Protocol Buffer migration path for the client-side of the Cosmos SDK.

Specifically, the client-side migration path primarily includes tx generation and signing, message construction and routing, in addition to CLI & REST handlers and business logic (i.e. queriers).

With this in mind, we will tackle the migration path via two main areas, txs and querying. However, this ADR solely focuses on transactions. Querying should be addressed in a future ADR, but it should build off of these proposals.

Based on detailed discussions ([#6030](#) and [#6078](#)), the original design for transactions was changed substantially from an `oneof` /JSON-signing approach to the approach described below.

## Decision

### Transactions

Since interface values are encoded with `google.protobuf.Any` in state (see [ADR 019](#)), `sdk.Msg` s are encoding with `Any` in transactions.

One of the main goals of using `Any` to encode interface values is to have a core set of types which is reused by apps so that clients can safely be compatible with as many chains as possible.

It is one of the goals of this specification to provide a flexible cross-chain transaction format that can serve a wide variety of use cases without breaking client compatibility.

In order to facilitate signing, transactions are separated into `TxBody`, which will be re-used by `SignDoc` below, and `signatures`:

```
// types/types.proto
package cosmos_sdk.v1;

message Tx {
 TxBody body = 1;
 AuthInfo auth_info = 2;
 // A list of signatures that matches the length and order of AuthInfo's
 signer_infos to
 // allow connecting signature meta information like public key and signing mode
 // by position.
 repeated bytes signatures = 3;
}

// A variant of Tx that pins the signer's exact binary representation of body and
// auth_info. This is used for signing, broadcasting and verification. The binary
// `serialize(tx: TxRaw)` is stored in Tendermint and the hash `sha256(serialize(tx:
TxRaw))` -
// becomes the "txhash", commonly used as the transaction ID.
message TxRaw {
 // A protobuf serialization of a TxBody that matches the representation in
 SignDoc.
 bytes body = 1;
 // A protobuf serialization of an AuthInfo that matches the representation in
 SignDoc.
 bytes auth_info = 2;
 // A list of signatures that matches the length and order of AuthInfo's
 signer_infos to
 // allow connecting signature meta information like public key and signing mode
 // by position.
 repeated bytes signatures = 3;
}

message TxBody {
 // A list of messages to be executed. The required signers of those messages
 define
 // the number and order of elements in AuthInfo's signer_infos and Tx's
 signatures.
 // Each required signer address is added to the list only the first time it
 occurs.
 //
 // By convention, the first required signer (usually from the first message) is
 referred
 // to as the primary signer and pays the fee for the whole transaction.
 repeated google.protobuf.Any messages = 1;
 string memo = 2;
 int64 timeout_height = 3;
```

```

 repeated google.protobuf.Any extension_options = 1023;
}

message AuthInfo {
 // This list defines the signing modes for the required signers. The number
 // and order of elements must match the required signers from TxBody's messages.
 // The first element is the primary signer and the one which pays the fee.
 repeated SignerInfo signer_infos = 1;
 // The fee can be calculated based on the cost of evaluating the body and doing
 // signature verification of the signers. This can be estimated via simulation.
 Fee fee = 2;
}

message SignerInfo {
 // The public key is optional for accounts that already exist in state. If
 // unset, the
 // verifier can use the required signer address for this position and lookup the
 // public key.
 google.protobuf.Any public_key = 1;
 // ModeInfo describes the signing mode of the signer and is a nested
 // structure to support nested multisig pubkey's
 ModeInfo mode_info = 2;
 // sequence is the sequence of the account, which describes the
 // number of committed transactions signed by a given address. It is used to
 prevent
 // replay attacks.
 uint64 sequence = 3;
}

message ModeInfo {
 oneof sum {
 Single single = 1;
 Multi multi = 2;
 }

 // Single is the mode info for a single signer. It is structured as a message
 // to allow for additional fields such as locale for SIGN_MODE_TEXTUAL in the
 future
 message Single {
 SignMode mode = 1;
 }

 // Multi is the mode info for a multisig public key
 message Multi {
 // bitarray specifies which keys within the multisig are signing
 CompactBitArray bitarray = 1;
 // mode_infos is the corresponding modes of the signers of the multisig
 // which could include nested multisig public keys
 repeated ModeInfo mode_infos = 2;
 }
}

```

```

enum SignMode {
 SIGN_MODE_UNSPECIFIED = 0;

 SIGN_MODE_DIRECT = 1;

 SIGN_MODE_TEXTUAL = 2;

 SIGN_MODE_LEGACY_AMINO_JSON = 127;
}

```

As will be discussed below, in order to include as much of the `Tx` as possible in the `SignDoc`, `SignerInfo` is separated from signatures so that only the raw signatures themselves live outside of what is signed over.

Because we are aiming for a flexible, extensible cross-chain transaction format, new transaction processing options should be added to `TxBody` as soon those use cases are discovered, even if they can't be implemented yet.

Because there is coordination overhead in this, `TxBody` includes an `extension_options` field which can be used for any transaction processing options that are not already covered. App developers should, nevertheless, attempt to upstream important improvements to `Tx`.

## **Signing**

All of the signing modes below aim to provide the following guarantees:

- **No Malleability:** `TxBody` and `AuthInfo` cannot change once the transaction is signed
- **Predictable Gas:** if I am signing a transaction where I am paying a fee, the final gas is fully dependent on what I am signing

These guarantees give the maximum amount confidence to message signers that manipulation of `Tx`'s by intermediaries can't result in any meaningful changes.

### **SIGN\_MODE\_DIRECT**

The "direct" signing behavior is to sign the raw `TxBody` bytes as broadcast over the wire. This has the advantages of:

- requiring the minimum additional client capabilities beyond a standard protocol buffers implementation
- leaving effectively zero holes for transaction malleability (i.e. there are no subtle differences between the signing and encoding formats which could potentially be exploited by an attacker)

Signatures are structured using the `SignDoc` below which reuses the serialization of `TxBody` and `AuthInfo` and only adds the fields which are needed for signatures:

```

// types/types.proto
message SignDoc {
 // A protobuf serialization of a TxBody that matches the representation in
 // TxRaw.
 bytes body = 1;
 // A protobuf serialization of an AuthInfo that matches the representation in
 // TxRaw.

```

```
 bytes auth_info = 2;
 string chain_id = 3;
 uint64 account_number = 4;
}
```

In order to sign in the default mode, clients take the following steps:

1. Serialize `TxBody` and `AuthInfo` using any valid protobuf implementation.
2. Create a `SignDoc` and serialize it using [ADR 027](#).
3. Sign the encoded `SignDoc` bytes.
4. Build a `TxRaw` and serialize it for broadcasting.

Signature verification is based on comparing the raw `TxBody` and `AuthInfo` bytes encoded in `TxRaw` not based on any "[canonicalization](#)" algorithm which creates added complexity for clients in addition to preventing some forms of upgradeability (to be addressed later in this document).

Signature verifiers do:

1. Deserialize a `TxRaw` and pull out `body` and `auth_info`.
2. Create a list of required signer addresses from the messages.
3. For each required signer:
  - o Pull account number and sequence from the state.
  - o Obtain the public key either from state or `AuthInfo`'s `signer_infos`.
  - o Create a `SignDoc` and serialize it using [ADR 027](#).
  - o Verify the signature at the same list position against the serialized `SignDoc`.

#### **SIGN\_MODE\_LEGACY\_AMINO**

In order to support legacy wallets and exchanges, Amino JSON will be temporarily supported transaction signing. Once wallets and exchanges have had a chance to upgrade to protobuf based signing, this option will be disabled. In the meantime, it is foreseen that disabling the current Amino signing would cause too much breakage to be feasible. Note that this is mainly a requirement of the Cosmos Hub and other chains may choose to disable Amino signing immediately.

Legacy clients will be able to sign a transaction using the current Amino JSON format and have it encoded to protobuf using the REST `/tx/encode` endpoint before broadcasting.

#### **SIGN\_MODE\_TEXTUAL**

As was discussed extensively in [#6078](#), there is a desire for a human-readable signing encoding, especially for hardware wallets like the [Ledger](#) which display transaction contents to users before signing. JSON was an attempt at this but falls short of the ideal.

`SIGN_MODE_TEXTUAL` is intended as a placeholder for a human-readable encoding which will replace Amino JSON. This new encoding should be even more focused on readability than JSON, possibly based on formatting strings like [MessageFormat](#).

In order to ensure that the new human-readable format does not suffer from transaction malleability issues, `SIGN_MODE_TEXTUAL` requires that the *human-readable bytes are concatenated with the raw `SignDoc`* to generate sign bytes.

Multiple human-readable formats (maybe even localized messages) may be supported by `SIGN_MODE_TEXTUAL` when it is implemented.

## Unknown Field Filtering

Unknown fields in protobuf messages should generally be rejected by transaction processors because:

- important data may be present in the unknown fields, that if ignored, will cause unexpected behavior for clients
- they present a malleability vulnerability where attackers can bloat tx size by adding random uninterpreted data to unsigned content (i.e. the master `Tx`, not `TxBody`)

There are also scenarios where we may choose to safely ignore unknown fields

(<https://github.com/cosmos/cosmos-sdk/issues/6078#issuecomment-624400188>) to provide graceful forwards compatibility with newer clients.

We propose that field numbers with bit 11 set (for most use cases this is the range of 1024-2047) be considered non-critical fields that can safely be ignored if unknown.

To handle this we will need a unknown field filter that:

- always rejects unknown fields in unsigned content (i.e. top-level `Tx` and unsigned parts of `AuthInfo` if present based on the signing mode)
- rejects unknown fields in all messages (including nested `Any`'s) other than fields with bit 11 set

This will likely need to be a custom protobuf parser pass that takes message bytes and `FileDescriptor`'s and returns a boolean result.

## Public Key Encoding

Public keys in the Cosmos SDK implement the `cryptotypes.PubKey` interface. We propose to use `Any` for protobuf encoding as we are doing with other interfaces (for example, in `BaseAccount.PubKey` and `SignerInfo.PublicKey`). The following public keys are implemented: secp256k1, secp256r1, ed25519 and legacy-multisignature.

Ex:

```
message PubKey {
 bytes key = 1;
}
```

`multisig.LegacyAminoPubKey` has an array of `Any`'s member to support any protobuf public key type.

Apps should only attempt to handle a registered set of public keys that they have tested. The provided signature verification ante handler decorators will enforce this.

## CLI & REST

Currently, the REST and CLI handlers encode and decode types and txs via Amino JSON encoding using a concrete Amino codec. Being that some of the types dealt with in the client can be interfaces, similar to how we described in [ADR 019](#), the client logic will now need to take a codec interface that knows not only how to handle all the types, but also knows how to generate transactions, signatures, and messages.

```
type AccountRetriever interface {
 GetAccount(clientCtx Context, addr sdk.AccAddress) (client.Account, error)
 GetAccountWithHeight(clientCtx Context, addr sdk.AccAddress) (client.Account,
```

```

int64, error)
 EnsureExists(clientCtx client.Context, addr sdk.AccAddress) error
 GetAccountNumberSequence(clientCtx client.Context, addr sdk.AccAddress) (uint64,
 uint64, error)
}

type Generator interface {
 NewTx() TxBuilder
 NewFee() ClientFee
 NewSignature() ClientSignature
 MarshalTx(tx types.Tx) ([]byte, error)
}

type TxBuilder interface {
 GetTx() sdk.Tx

 SetMsgs(...sdk.Msg) error
 GetSignatures() []sdk.Signature
 SetSignatures(...sdk.Signature)
 GetFee() sdk.Fee
 SetFee(sdk.Fee)
 GetMemo() string
 SetMemo(string)
}

```

We then update `Context` to have new fields: `Codec`, `TxGenerator`, and `AccountRetriever`, and we update  `AppModuleBasic.GetTxCmd` to take a `Context` which should have all of these fields pre-populated.

Each client method should then use one of the `Init` methods to re-initialize the pre-populated `Context`. `tx.GenerateOrBroadcastTx` can be used to generate or broadcast a transaction. For example:

```

import "github.com/spf13/cobra"
import "github.com/cosmos/cosmos-sdk/client"
import "github.com/cosmos/cosmos-sdk/client/tx"

func NewCmdDoSomething(clientCtx client.Context) *cobra.Command {
 return &cobra.Command{
 RunE: func(cmd *cobra.Command, args []string) error {
 clientCtx := ctx.InitWithInput(cmd.InOrStdin())
 msg := NewSomeMsg{...}
 tx.GenerateOrBroadcastTx(clientCtx, msg)
 },
 }
}

```

## Future Improvements

`SIGN_MODE_TEXTUAL` specification

A concrete specification and implementation of `SIGN_MODE_TEXTUAL` is intended as a near-term future improvement so that the ledger app and other wallets can gracefully transition away from Amino JSON.

### `SIGN_MODE_DIRECT_AUX`

(\*Documented as option (3) in <https://github.com/cosmos/cosmos-sdk/issues/6078#issuecomment-628026933>)

We could add a mode `SIGN_MODE_DIRECT_AUX` to support scenarios where multiple signatures are being gathered into a single transaction but the message composer does not yet know which signatures will be included in the final transaction. For instance, I may have a 3/5 multisig wallet and want to send a `TxBody` to all 5 signers to see who signs first. As soon as I have 3 signatures then I will go ahead and build the full transaction.

With `SIGN_MODE_DIRECT`, each signer needs to sign the full `AuthInfo` which includes the full list of all signers and their signing modes, making the above scenario very hard.

`SIGN_MODE_DIRECT_AUX` would allow "auxiliary" signers to create their signature using only `TxBody` and their own `PublicKey`. This allows the full list of signers in `AuthInfo` to be delayed until signatures have been collected.

An "auxiliary" signer is any signer besides the primary signer who is paying the fee. For the primary signer, the full `AuthInfo` is actually needed to calculate gas and fees because that is dependent on how many signers and which key types and signing modes they are using. Auxiliary signers, however, do not need to worry about fees or gas and thus can just sign `TxBody`.

To generate a signature in `SIGN_MODE_DIRECT_AUX` these steps would be followed:

1. Encode `SignDocAux` (with the same requirement that fields must be serialized in order):

```
// types/types.proto
message SignDocAux {
 bytes body_bytes = 1;
 // PublicKey is included in SignDocAux :
 // 1. as a special case for multisig public keys. For multisig public
 // keys,
 // the signer should use the top-level multisig public key they are
 // signing
 // against, not their own public key. This is to prevent against a form
 // of malleability where a signature could be taken out of context of the
 // multisig key that was intended to be signed for
 // 2. to guard against scenario where configuration information is
 // encoded
 // in public keys (it has been proposed) such that two keys can generate
 // the same signature but have different security properties
 //
 // By including it here, the composer of AuthInfo cannot reference the
 // a public key variant the signer did not intend to use
 PublicKey public_key = 2;
 string chain_id = 3;
 uint64 account_number = 4;
}
```

2. Sign the encoded `SignDocAux` bytes
3. Send their signature and `SignerInfo` to primary signer who will then sign and broadcast the final transaction (with `SIGN_MODE_DIRECT` and `AuthInfo` added) once enough signatures have been collected

#### **`SIGN_MODE_DIRECT_RELAXED`**

(Documented as option (1)(a) in <https://github.com/cosmos/cosmos-sdk/issues/6078#issuecomment-628026933>)

This is a variation of `SIGN_MODE_DIRECT` where multiple signers wouldn't need to coordinate public keys and signing modes in advance. It would involve an alternate `SignDoc` similar to `SignDocAux` above with fee. This could be added in the future if client developers found the burden of collecting public keys and modes in advance too burdensome.

## **Consequences**

### **Positive**

- Significant performance gains.
- Supports backward and forward type compatibility.
- Better support for cross-language clients.
- Multiple signing modes allow for greater protocol evolution

### **Negative**

- `google.protobuf.Any` type URLs increase transaction size although the effect may be negligible or compression may be able to mitigate it.

### **Neutral**

## **References**

# **ADR 021: Protocol Buffer Query Encoding**

### **Changelog**

- 2020 March 27: Initial Draft

### **Status**

Accepted

### **Context**

This ADR is a continuation of the motivation, design, and context established in [ADR 019](#) and [ADR 020](#), namely, we aim to design the Protocol Buffer migration path for the client-side of the Cosmos SDK.

This ADR continues from [ADR 020](#) to specify the encoding of queries.

### **Decision**

## Custom Query Definition

Modules define custom queries through a protocol buffers `service` definition. These `service` definitions are generally associated with and used by the GRPC protocol. However, the protocol buffers specification indicates that they can be used more generically by any request/response protocol that uses protocol buffer encoding. Thus, we can use `service` definitions for specifying custom ABCI queries and even reuse a substantial amount of the GRPC infrastructure.

Each module with custom queries should define a service canonically named `Query`:

```
// x/bank/types/types.proto

service Query {
 rpc QueryBalance(QueryBalanceParams) returns (cosmos_sdk.v1.Coin) {}
 rpc QueryAllBalances(QueryAllBalancesParams) returns (QueryAllBalancesResponse) {}
}
```

## Handling of Interface Types

Modules that use interface types and need true polymorphism generally force a `oneof` up to the app-level that provides the set of concrete implementations of that interface that the app supports. While app's are welcome to do the same for queries and implement an app-level query service, it is recommended that modules provide query methods that expose these interfaces via `google.protobuf.Any`. There is a concern on the transaction level that the overhead of `Any` is too high to justify its usage. However for queries this is not a concern, and providing generic module-level queries that use `Any` does not preclude apps from also providing app-level queries that return use the app-level `oneof`s.

A hypothetical example for the `gov` module would look something like:

```
// x/gov/types/types.proto

import "google/protobuf/any.proto";

service Query {
 rpc GetProposal(GetProposalParams) returns (AnyProposal) {}
}

message AnyProposal {
 ProposalBase base = 1;
 google.protobuf.Any content = 2;
}
```

## Custom Query Implementation

In order to implement the query service, we can reuse the existing [gogo.protobuf](#) grpc plugin, which for a service named `Query` generates an interface named `QueryServer` as below:

```
type QueryServer interface {
 QueryBalance(context.Context, *QueryBalanceParams) (*types.Coin, error)
```

```

 QueryAllBalances(context.Context, *QueryAllBalancesParams)
 (*QueryAllBalancesResponse, error)
}

```

The custom queries for our module are implemented by implementing this interface.

The first parameter in this generated interface is a generic `context.Context`, whereas querier methods generally need an instance of `sdk.Context` to read from the store. Since arbitrary values can be attached to `context.Context` using the `WithValue` and `Value` methods, the Cosmos SDK should provide a function `sdk.UnwrapSDKContext` to retrieve the `sdk.Context` from the provided `context.Context`.

An example implementation of `QueryBalance` for the bank module as above would look something like:

```

type Querier struct {
 Keeper
}

func (q Querier) QueryBalance(ctx context.Context, params *types.QueryBalanceParams)
(*sdk.Coin, error) {
 balance := q.GetBalance(sdk.UnwrapSDKContext(ctx), params.Address, params.Denom)
 return &balance, nil
}

```

## Custom Query Registration and Routing

Query server implementations as above would be registered with `AppModule`'s using a new method `RegisterQueryService(grpc.Server)` which could be implemented simply as below:

```

// x/bank/module.go
func (am AppModule) RegisterQueryService(server grpc.Server) {
 types.RegisterQueryServer(server, keeper.Querier{am.keeper})
}

```

Underneath the hood, a new method `RegisterService(sd *grpc.ServiceDesc, handler interface{})` will be added to the existing `baseapp.QueryRouter` to add the queries to the custom query routing table (with the routing method being described below). The signature for this method matches the existing `RegisterServer` method on the GRPC `Server` type where `handler` is the custom query server implementation described above.

GRPC-like requests are routed by the service name (ex. `cosmos_sdk.x.bank.v1.Query`) and method name (ex. `QueryBalance`) combined with `/`'s to form a full method name (ex. `/cosmos_sdk.x.bank.v1.Query/QueryBalance`). This gets translated into an ABCI query as `custom/cosmos_sdk.x.bank.v1.Query/QueryBalance`. Service handlers registered with `QueryRouter.RegisterService` will be routed this way.

Beyond the method name, GRPC requests carry a protobuf encoded payload, which maps naturally to `RequestQuery.Data`, and receive a protobuf encoded response or error. Thus there is a quite natural mapping of GRPC-like rpc methods to the existing `sdk.Query` and `QueryRouter` infrastructure.

This basic specification allows us to reuse protocol buffer `service` definitions for ABCI custom queries substantially reducing the need for manual decoding and encoding in query methods.

## GRPC Protocol Support

In addition to providing an ABCI query pathway, we can easily provide a GRPC proxy server that routes requests in the GRPC protocol to ABCI query requests under the hood. In this way, clients could use their host languages' existing GRPC implementations to make direct queries against Cosmos SDK app's using these `service` definitions. In order for this server to work, the `QueryRouter` on `BaseApp` will need to expose the service handlers registered with `QueryRouter.RegisterService` to the proxy server implementation. Nodes could launch the proxy server on a separate port in the same process as the ABCI app with a command-line flag.

## REST Queries and Swagger Generation

[grpc-gateway](#) is a project that translates REST calls into GRPC calls using special annotations on service methods. Modules that want to expose REST queries should add `google.api.http` annotations to their `rpc` methods as in this example below.

```
// x/bank/types/types.proto

service Query {
 rpc QueryBalance(QueryBalanceParams) returns (cosmos_sdk.v1.Coin) {
 option (google.api.http) = {
 get: "/x/bank/v1/balance/{address}/{denom}"
 };
 }
 rpc QueryAllBalances(QueryAllBalancesParams) returns (QueryAllBalancesResponse) {
 option (google.api.http) = {
 get: "/x/bank/v1/balances/{address}"
 };
 }
}
```

grpc-gateway will work directly against the GRPC proxy described above which will translate requests to ABCI queries under the hood. grpc-gateway can also generate Swagger definitions automatically.

In the current implementation of REST queries, each module needs to implement REST queries manually in addition to ABCI querier methods. Using the grpc-gateway approach, there will be no need to generate separate REST query handlers, just query servers as described above as grpc-gateway handles the translation of protobuf to REST as well as Swagger definitions.

The Cosmos SDK should provide CLI commands for apps to start GRPC gateway either in a separate process or the same process as the ABCI app, as well as provide a command for generating grpc-gateway proxy `.proto` files and the `swagger.json` file.

## Client Usage

The gogo protobuf grpc plugin generates client interfaces in addition to server interfaces. For the `Query` service defined above we would get a `QueryClient` interface like:

```

type QueryClient interface {
 QueryBalance(ctx context.Context, in *QueryBalanceParams, opts
...grpc.CallOption) (*types.Coin, error)
 QueryAllBalances(ctx context.Context, in *QueryAllBalancesParams, opts
...grpc.CallOption) (*QueryAllBalancesResponse, error)
}

```

Via a small patch to gogo protobuf ([gogo/protobuf#675](#)) we have tweaked the grpc codegen to use an interface rather than concrete type for the generated client struct. This allows us to also reuse the GRPC infrastructure for ABCI client queries.

1Context will receive a new method `QueryConn` that returns a `ClientConn`` that routes calls to ABCI queries

Clients (such as CLI methods) will then be able to call query methods like this:

```

clientCtx := client.NewContext()
queryClient := types.NewQueryClient(clientCtx.QueryConn())
params := &types.QueryBalanceParams{addr, denom}
result, err := queryClient.QueryBalance(gocontext.Background(), params)

```

## Testing

Tests would be able to create a query client directly from keeper and `sdk.Context` references using a `QueryServerTestHelper` as below:

```

queryHelper := baseapp.NewQueryServerTestHelper(ctx)
types.RegisterQueryServer(queryHelper, keeper.Querier{app.BankKeeper})
queryClient := types.NewQueryClient(queryHelper)

```

## Future Improvements

### Consequences

#### Positive

- greatly simplified querier implementation (no manual encoding/decoding)
- easy query client generation (can use existing grpc and swagger tools)
- no need for REST query implementations
- type safe query methods (generated via grpc plugin)
- going forward, there will be less breakage of query methods because of the backwards compatibility guarantees provided by buf

#### Negative

- all clients using the existing ABCI/REST queries will need to be refactored for both the new GRPC/REST query paths as well as protobuf/proto-json encoded data, but this is more or less unavoidable in the protobuf refactoring

#### Neutral

## References

# ADR 022: Custom BaseApp panic handling

## Changelog

- 2020 Apr 24: Initial Draft
- 2021 Sep 14: Superseded by ADR-045

## Status

SUPERSEDED by ADR-045

## Context

The current implementation of BaseApp does not allow developers to write custom error handlers during panic recovery `runTx()` method. We think that this method can be more flexible and can give Cosmos SDK users more options for customizations without the need to rewrite whole BaseApp. Also there's one special case for `sdk.ErrorOutOfGas` error handling, that case might be handled in a "standard" way (middleware) alongside the others.

We propose middleware-solution, which could help developers implement the following cases:

- add external logging (let's say sending reports to external services like [Sentry](#));
- call panic for specific error cases;

It will also make `OutOfGas` case and `default` case one of the middlewares. `Default` case wraps recovery object to an error and logs it ([example middleware implementation](#)).

Our project has a sidecar service running alongside the blockchain node (smart contracts virtual machine). It is essential that node <-> sidecar connectivity stays stable for TXs processing. So when the communication breaks we need to crash the node and reboot it once the problem is solved. That behaviour makes node's state machine execution deterministic. As all keeper panics are caught by runTx's `defer()` handler, we have to adjust the BaseApp code in order to customize it.

## Decision

### Design

#### Overview

Instead of hardcoding custom error handling into BaseApp we suggest using set of middlewares which can be customized externally and will allow developers use as many custom error handlers as they want.

Implementation with tests can be found [here](#).

#### Implementation details

##### Recovery handler

New `RecoveryHandler` type added. `recoveryObj` input argument is an object returned by the standard Go function `recover()` from the `builtin` package.

```
type RecoveryHandler func(recoveryObj interface{}) error
```

Handler should type assert (or other methods) an object to define if object should be handled. `nil` should be returned if input object can't be handled by that `RecoveryHandler` (not a handler's target type). Not `nil` error should be returned if input object was handled and middleware chain execution should be stopped.

An example:

```
func exampleErrHandler(recoveryObj interface{}) error {
 err, ok := recoveryObj.(error)
 if !ok { return nil }

 if someSpecificError.Is(err) {
 panic(customPanicMsg)
 } else {
 return nil
 }
}
```

This example breaks the application execution, but it also might enrich the error's context like the `OutOfGas` handler.

### Recovery middleware

We also add a middleware type (decorator). That function type wraps `RecoveryHandler` and returns the next middleware in execution chain and handler's `error`. Type is used to separate actual `recovery()` object handling from middleware chain processing.

```
type recoveryMiddleware func(recoveryObj interface{}) (recoveryMiddleware, error)

func newRecoveryMiddleware(handler RecoveryHandler, next recoveryMiddleware) recoveryMiddleware {
 return func(recoveryObj interface{}) (recoveryMiddleware, error) {
 if err := handler(recoveryObj); err != nil {
 return nil, err
 }
 return next, nil
 }
}
```

Function receives a `recoveryObj` object and returns:

- (`next recoveryMiddleware, nil`) if object wasn't handled (not a target type) by `RecoveryHandler`;
- (`nil, not nil error`) if input object was handled and other middlewares in the chain should not be executed;
- (`nil, nil`) in case of invalid behavior. Panic recovery might not have been properly handled; this can be avoided by always using a `default` as a rightmost middleware in the chain (always returns an `error`');

`OutOfGas` middleware example:

```
func newOutOfGasRecoveryMiddleware(gasWanted uint64, ctx sdk.Context, next
recoveryMiddleware) recoveryMiddleware {
 handler := func(recoveryObj interface{}) error {
 err, ok := recoveryObj.(sdk.ErrorOutOfGas)
 if !ok { return nil }

 return errorsmod.Wrap(
 sdkerrors.ErrOutOfGas, fmt.Sprintf(
 "out of gas in location: %v; gasWanted: %d, gasUsed: %d",
 err.Descriptor, gasWanted, ctx.GasMeter().GasConsumed(),
),
)
}

return newRecoveryMiddleware(handler, next)
}
```

`Default` middleware example:

```
func newDefaultRecoveryMiddleware() recoveryMiddleware {
 handler := func(recoveryObj interface{}) error {
 return errorsmod.Wrap(
 sdkerrors.ErrPanic, fmt.Sprintf("recovered: %v\nstack:\n%v",
recoveryObj, string(debug.Stack())),
)
}

return newRecoveryMiddleware(handler, nil)
}
```

## Recovery processing

Basic chain of middlewares processing would look like:

```
func processRecovery(recoveryObj interface{}, middleware recoveryMiddleware) error {
 if middleware == nil { return nil }

 next, err := middleware(recoveryObj)
 if err != nil { return err }
 if next == nil { return nil }

 return processRecovery(recoveryObj, next)
}
```

That way we can create a middleware chain which is executed from left to right, the rightmost middleware is a `default` handler which must return an `error`.

## BaseApp changes

The `default` middleware chain must exist in a `BaseApp` object. `Baseapp` modifications:

```

type BaseApp struct {
 // ...
 runTxRecoveryMiddleware recoveryMiddleware
}

func NewBaseApp(...) {
 // ...
 app.runTxRecoveryMiddleware = newDefaultRecoveryMiddleware()
}

func (app *BaseApp) runTx(...) {
 // ...
 defer func() {
 if r := recover(); r != nil {
 recoveryMW := newOutOfGasRecoveryMiddleware(gasWanted, ctx,
app.runTxRecoveryMiddleware)
 err, result = processRecovery(r, recoveryMW), nil
 }
 }

 gInfo = sdk.GasInfo{GasWanted: gasWanted, GasUsed:
ctx.GasMeter().GasConsumed()}
 }()
 // ...
}

```

Developers can add their custom `RecoveryHandler`s by providing `AddRunTxRecoveryHandler` as a `BaseApp` option parameter to the `NewBaseapp` constructor:

```

func (app *BaseApp) AddRunTxRecoveryHandler(handlers ...RecoveryHandler) {
 for _, h := range handlers {
 app.runTxRecoveryMiddleware = newRecoveryMiddleware(h,
app.runTxRecoveryMiddleware)
 }
}

```

This method would prepend handlers to an existing chain.

## Consequences

### Positive

- Developers of Cosmos SDK based projects can add custom panic handlers to:
  - add error context for custom panic sources (panic inside of custom keepers);
  - emit `panic()` : passthrough recovery object to the Tendermint core;
  - other necessary handling;
- Developers can use standard Cosmos SDK `BaseApp` implementation, rather than rewriting it in their projects;
- Proposed solution doesn't break the current "standard" `runTx()` flow;

### Negative

- Introduces changes to the execution model design.

## Neutral

- `OutOfGas` error handler becomes one of the middlewares;
- Default panic handler becomes one of the middlewares;

## References

- [PR-6053 with proposed solution](#)
- [Similar solution, ADR-010 Modular AnteHandler](#)

# ADR 023: Protocol Buffer Naming and Versioning Conventions

## Changelog

- 2020 April 27: Initial Draft
- 2020 August 5: Update guidelines

## Status

Accepted

## Context

Protocol Buffers provide a basic [style guide](#) and [Buf](#) builds upon that. To the extent possible, we want to follow industry accepted guidelines and wisdom for the effective usage of protobuf, deviating from those only when there is clear rationale for our use case.

### Adoption of `Any`

The adoption of `google.protobuf.Any` as the recommended approach for encoding interface types (as opposed to `oneof`) makes package naming a central part of the encoding as fully-qualified message names now appear in encoded messages.

### Current Directory Organization

Thus far we have mostly followed [Buf's DEFAULT](#) recommendations, with the minor deviation of disabling [PACKAGE\\_DIRECTORY\\_MATCH](#) which although being convenient for developing code comes with the warning from Buf that:

*you will have a very bad time with many Protobuf plugins across various languages if you do not do this*

### Adoption of gRPC Queries

In [ADR 021](#), gRPC was adopted for Protobuf native queries. The full gRPC service path thus becomes a key part of ABCI query path. In the future, gRPC queries may be allowed from within persistent scripts by technologies such as CosmWasm and these query routes would be stored within script binaries.

## Decision

The goal of this ADR is to provide thoughtful naming conventions that:

- encourage a good user experience for when users interact directly with .proto files and fully-qualified protobuf names
- balance conciseness against the possibility of either over-optimizing (making names too short and cryptic) or under-optimizing (just accepting bloated names with lots of redundant information)

These guidelines are meant to act as a style guide for both the Cosmos SDK and third-party modules.

As a starting point, we should adopt all of the [DEFAULT](#) checkers in [Buf's](#) including [PACKAGE\\_DIRECTORY\\_MATCH](#), except:

- [PACKAGE\\_VERSION\\_SUFFIX](#)
- [SERVICE\\_SUFFIX](#)

Further guidelines to be described below.

## Principles

### Concise and Descriptive Names

Names should be descriptive enough to convey their meaning and distinguish them from other names.

Given that we are using fully-qualified names within `google.protobuf.Any` as well as within gRPC query routes, we should aim to keep names concise, without going overboard. The general rule of thumb should be if a shorter name would convey more or else the same thing, pick the shorter name.

For instance, `cosmos.bank.MsgSend` (19 bytes) conveys roughly the same information as `cosmos_sdk.x.bank.v1.MsgSend` (28 bytes) but is more concise.

Such conciseness makes names both more pleasant to work with and take up less space within transactions and on the wire.

We should also resist the temptation to over-optimize, by making names cryptically short with abbreviations. For instance, we shouldn't try to reduce `cosmos.bank.MsgSend` to `csm.bk.MSnd` just to save a few bytes.

The goal is to make names **concise but not cryptic**.

### Names are for Clients First

Package and type names should be chosen for the benefit of users, not necessarily because of legacy concerns related to the go code-base.

### Plan for Longevity

In the interests of long-term support, we should plan on the names we do choose to be in usage for a long time, so now is the opportunity to make the best choices for the future.

## Versioning

### Guidelines on Stable Package Versions

In general, schema evolution is the way to update protobuf schemas. That means that new fields, messages, and RPC methods are *added* to existing schemas and old fields, messages and RPC methods are maintained as long as possible.

Breaking things is often unacceptable in a blockchain scenario. For instance, immutable smart contracts may depend on certain data schemas on the host chain. If the host chain breaks those schemas, the smart contract may be irreparably broken. Even when things can be fixed (for instance in client software), this often comes at a high cost.

Instead of breaking things, we should make every effort to evolve schemas rather than just breaking them. [Buf](#) breaking change detection should be used on all stable (non-alpha or beta) packages to prevent such breakage.

With that in mind, different stable versions (i.e. `v1` or `v2`) of a package should more or less be considered different packages and this should be last resort approach for upgrading protobuf schemas. Scenarios where creating a `v2` may make sense are:

- we want to create a new module with similar functionality to an existing module and adding `v2` is the most natural way to do this. In that case, there are really just two different, but similar modules with different APIs.
- we want to add a new revamped API for an existing module and it's just too cumbersome to add it to the existing package, so putting it in `v2` is cleaner for users. In this case, care should be made to not deprecate support for `v1` if it is actively used in immutable smart contracts.

### Guidelines on unstable (alpha and beta) package versions

The following guidelines are recommended for marking packages as alpha or beta:

- marking something as `alpha` or `beta` should be a last resort and just putting something in the stable package (i.e. `v1` or `v2`) should be preferred
- a package *should* be marked as `alpha` *if and only if* there are active discussions to remove or significantly alter the package in the near future
- a package *should* be marked as `beta` *if and only if* there is an active discussion to significantly refactor/rework the functionality in the near future but not remove it
- modules *can and should* have types in both stable (i.e. `v1` or `v2`) and unstable (`alpha` or `beta`) packages.

`alpha` and `beta` should not be used to avoid responsibility for maintaining compatibility. Whenever code is released into the wild, especially on a blockchain, there is a high cost to changing things. In some cases, for instance with immutable smart contracts, a breaking change may be impossible to fix.

When marking something as `alpha` or `beta`, maintainers should ask the questions:

- what is the cost of asking others to change their code vs the benefit of us maintaining the optionality to change it?
- what is the plan for moving this to `v1` and how will that affect users?

`alpha` or `beta` should really be used to communicate "changes are planned".

As a case study, gRPC reflection is in the package `grpc.reflection.v1alpha`. It hasn't been changed since 2017 and it is now used in other widely used software like gRPCcurl. Some folks probably use it in production services and so if they actually went and changed the package to `grpc.reflection.v1`, some software would break and they probably don't want to do that... So now the `v1alpha` package is more or less the de-facto `v1`. Let's not do that.

The following are guidelines for working with non-stable packages:

- [Buf's recommended version suffix](#) (ex. `v1alpha1`) *should* be used for non-stable packages

- non-stable packages should generally be excluded from breaking change detection
- immutable smart contract modules (i.e. CosmWasm) *should* block smart contracts/persistent scripts from interacting with `alpha` / `beta` packages

### Omit v1 suffix

Instead of using [Buf's recommended version suffix](#), we can omit `v1` for packages that don't actually have a second version. This allows for more concise names for common use cases like `cosmos.bank.Send`. Packages that do have a second or third version can indicate that with `.v2` or `.v3`.

## Package Naming

### Adopt a short, unique top-level package name

Top-level packages should adopt a short name that is known to not collide with other names in common usage within the Cosmos ecosystem. In the near future, a registry should be created to reserve and index top-level package names used within the Cosmos ecosystem. Because the Cosmos SDK is intended to provide the top-level types for the Cosmos project, the top-level package name `cosmos` is recommended for usage within the Cosmos SDK instead of the longer `cosmos_sdk`. [ICS](#) specifications could consider a short top-level package like `ics23` based upon the standard number.

### Limit sub-package depth

Sub-package depth should be increased with caution. Generally a single sub-package is needed for a module or a library. Even though `x` or `modules` is used in source code to denote modules, this is often unnecessary for .proto files as modules are the primary thing sub-packages are used for. Only items which are known to be used infrequently should have deep sub-package depths.

For the Cosmos SDK, it is recommended that we simply write `cosmos.bank`, `cosmos.gov`, etc. rather than `cosmos.x.bank`. In practice, most non-module types can go straight in the `cosmos` package or we can introduce a `cosmos.base` package if needed. Note that this naming *will not* change go package names, i.e. the `cosmos.bank` protobuf package will still live in `x/bank`.

## Message Naming

Message type names should be as concise possible without losing clarity. `sdk.Msg` types which are used in transactions will retain the `Msg` prefix as that provides helpful context.

## Service and RPC Naming

[ADR 021](#) specifies that modules should implement a gRPC query service. We should consider the principle of conciseness for query service and RPC names as these may be called from persistent script modules such as CosmWasm. Also, users may use these query paths from tools like [gRPCurl](#). As an example, we can shorten `/cosmos_sdk.x.bank.v1.QueryService/QueryBalance` to `/cosmos.bank.Query/Balance` without losing much useful information.

RPC request and response types *should* follow the `ServiceNameMethodNameRequest` / `ServiceNameMethodNameResponse` naming convention. i.e. for an RPC method named `Balance` on the `Query` service, the request and response types would be `QueryBalanceRequest` and `QueryBalanceResponse`. This will be more self-explanatory than `BalanceRequest` and `BalanceResponse`.

### Use just `Query` for the query service

Instead of [Buf's default service suffix recommendation](#), we should simply use the shorter `Query` for query services.

For other types of gRPC services, we should consider sticking with Buf's default recommendation.

#### Omit `Get` and `Query` from query service RPC names

`Get` and `Query` should be omitted from `Query` service names because they are redundant in the fully-qualified name. For instance, `/cosmos.bank.Query/QueryBalance` just says `Query` twice without any new information.

## Future Improvements

A registry of top-level package names should be created to coordinate naming across the ecosystem, prevent collisions, and also help developers discover useful schemas. A simple starting point would be a git repository with community-based governance.

## Consequences

### Positive

- names will be more concise and easier to read and type
- all transactions using `Any` will be at shorter (`_sdk.x` and `.v1` will be removed)
- `.proto` file imports will be more standard (without `"third_party/proto"` in the path)
- code generation will be easier for clients because `.proto` files will be in a single `proto/` directory which can be copied rather than scattered throughout the Cosmos SDK

### Negative

### Neutral

- `.proto` files will need to be reorganized and refactored
- some modules may need to be marked as alpha or beta

## References

# ADR 024: Coin Metadata

## Changelog

- 05/19/2020: Initial draft

## Status

ACCEPTED

## Context

Assets in the Cosmos SDK are represented via a `Coin` type that consists of an `amount` and a `denom`, where the `amount` can be any arbitrarily large or small value. In addition, the Cosmos SDK uses an account-based model where there are two types of primary accounts -- basic accounts and module accounts. All account types have a set of balances that are composed of `Coin`s. The `x/bank` module

keeps track of all balances for all accounts and also keeps track of the total supply of balances in an application.

With regards to a balance `amount`, the Cosmos SDK assumes a static and fixed unit of denomination, regardless of the denomination itself. In other words, clients and apps built atop a Cosmos-SDK-based chain may choose to define and use arbitrary units of denomination to provide a richer UX, however, by the time a tx or operation reaches the Cosmos SDK state machine, the `amount` is treated as a single unit. For example, for the Cosmos Hub (Gaia), clients assume  $1 \text{ ATOM} = 10^6 \text{ uatom}$ , and so all txs and operations in the Cosmos SDK work off of units of  $10^6$ .

This clearly provides a poor and limited UX especially as interoperability of networks increases and as a result the total amount of asset types increases. We propose to have `x/bank` additionally keep track of metadata per `denom` in order to help clients, wallet providers, and explorers improve their UX and remove the requirement for making any assumptions on the unit of denomination.

## Decision

The `x/bank` module will be updated to store and index metadata by `denom`, specifically the "base" or smallest unit -- the unit the Cosmos SDK state-machine works with.

Metadata may also include a non-zero length list of denominations. Each entry contains the name of the denomination `denom`, the exponent to the base and a list of aliases. An entry is to be interpreted as `1 denom = 10^exponent base_denom` (e.g. `1 ETH = 10^18 wei` and `1 uatom = 10^0 uatom`).

There are two denominations that are of high importance for clients: the `base`, which is the smallest possible unit and the `display`, which is the unit that is commonly referred to in human communication and on exchanges. The values in those fields link to an entry in the list of denominations.

The list in `denom_units` and the `display` entry may be changed via governance.

As a result, we can define the type as follows:

```
message DenomUnit {
 string denom = 1;
 uint32 exponent = 2;
 repeated string aliases = 3;
}

message Metadata {
 string description = 1;
 repeated DenomUnit denom_units = 2;
 string base = 3;
 string display = 4;
}
```

As an example, the ATOM's metadata can be defined as follows:

```
{
 "name": "atom",
 "description": "The native staking token of the Cosmos Hub.",
 "denom_units": [
```

```
{
 {
 "denom": "uatom",
 "exponent": 0,
 "aliases": [
 "microatom"
],
 },
 {
 "denom": "matom",
 "exponent": 3,
 "aliases": [
 "milliatom"
]
 },
 {
 "denom": "atom",
 "exponent": 6,
 }
],
"base": "uatom",
"display": "atom",
}
```

Given the above metadata, a client may infer the following things:

- $4.3\text{atom} = 4.3 * (10^6) = 4,300,000\text{uatom}$
- The string "atom" can be used as a display name in a list of tokens.
- The balance 4300000 can be displayed as 4,300,000uatom or 4,300matom or 4.3atom. The `display` denomination 4.3atom is a good default if the authors of the client don't make an explicit decision to choose a different representation.

A client should be able to query for metadata by denom both via the CLI and REST interfaces. In addition, we will add handlers to these interfaces to convert from any unit to another given unit, as the base framework for this already exists in the Cosmos SDK.

Finally, we need to ensure metadata exists in the `GenesisState` of the `x/bank` module which is also indexed by the `base denom`.

```
type GenesisState struct {
 SendEnabled bool `json:"send_enabled" yaml:"send_enabled"`
 Balances []Balance `json:"balances" yaml:"balances"`
 Supply sdk.Coins `json:"supply" yaml:"supply"`
 DenomMetadata []Metadata `json:"denom_metadata" yaml:"denom_metadata"`
}
```

## Future Work

In order for clients to avoid having to convert assets to the base denomination -- either manually or via an endpoint, we may consider supporting automatic conversion of a given unit input.

## Consequences

### Positive

- Provides clients, wallet providers and block explorers with additional data on asset denomination to improve UX and remove any need to make assumptions on denomination units.

### Negative

- A small amount of required additional storage in the `x/bank` module. The amount of additional storage should be minimal as the amount of total assets should not be large.

### Neutral

## References

# ADR 027: Deterministic Protobuf Serialization

## Changelog

- 2020-08-07: Initial Draft
- 2020-09-01: Further clarify rules

## Status

Proposed

## Abstract

Fully deterministic structure serialization, which works across many languages and clients, is needed when signing messages. We need to be sure that whenever we serialize a data structure, no matter in which supported language, the raw bytes will stay the same. [Protobuf](#) serialization is not bijective (i.e. there exist a practically unlimited number of valid binary representations for a given protobuf document)<sup>1</sup>.

This document describes a deterministic serialization scheme for a subset of protobuf documents, that covers this use case but can be reused in other cases as well.

## Context

For signature verification in Cosmos SDK, the signer and verifier need to agree on the same serialization of a `SignDoc` as defined in [ADR-020](#) without transmitting the serialization.

Currently, for block signatures we are using a workaround: we create a new `TxRaw` instance (as defined in [adr-020-protobuf-transaction-encoding](#)) by converting all `Tx` fields to bytes on the client side. This adds an additional manual step when sending and signing transactions.

## Decision

The following encoding scheme is to be used by other ADRs, and in particular for `SignDoc` serialization.

## Specification

## Scope

This ADR defines a protobuf3 serializer. The output is a valid protobuf serialization, such that every protobuf parser can parse it.

No maps are supported in version 1 due to the complexity of defining a deterministic serialization. This might change in future. Implementations must reject documents containing maps as invalid input.

## Background - Protobuf3 Encoding

Most numeric types in protobuf3 are encoded as [varints](#). Varints are at most 10 bytes, and since each varint byte has 7 bits of data, varints are a representation of `uint70` (70-bit unsigned integer). When encoding, numeric values are casted from their base type to `uint70`, and when decoding, the parsed `uint70` is casted to the appropriate numeric type.

The maximum valid value for a varint that complies with protobuf3 is `FF FF FF FF FF FF FF FF FF 7F` (i.e.  $2^{**70} - 1$ ). If the field type is `{,u,s}int64`, the highest 6 bits of the 70 are dropped during decoding, introducing 6 bits of malleability. If the field type is `{,u,s}int32`, the highest 38 bits of the 70 are dropped during decoding, introducing 38 bits of malleability.

Among other sources of non-determinism, this ADR eliminates the possibility of encoding malleability.

## Serialization rules

The serialization is based on the [protobuf3 encoding](#) with the following additions:

1. Fields must be serialized only once in ascending order
2. Extra fields or any extra data must not be added
3. [Default values](#) must be omitted
4. `repeated` fields of scalar numeric types must use [packed encoding](#)
5. Varint encoding must not be longer than needed:
  - o No trailing zero bytes (in little endian, i.e. no leading zeroes in big endian). Per rule 3 above, the default value of `0` must be omitted, so this rule does not apply in such cases.
  - o The maximum value for a varint must be `FF FF FF FF FF FF FF FF FF 01`. In other words, when decoded, the highest 6 bits of the 70-bit unsigned integer must be `0`. (10-byte varints are 10 groups of 7 bits, i.e. 70 bits, of which only the lowest 70-6=64 are useful.)
  - o The maximum value for 32-bit values in varint encoding must be `FF FF FF FF 0F` with one exception (below). In other words, when decoded, the highest 38 bits of the 70-bit unsigned integer must be `0`.
    - The one exception to the above is *negative* `int32`, which must be encoded using the full 10 bytes for sign extension<sup>2</sup>.
  - o The maximum value for Boolean values in varint encoding must be `01` (i.e. it must be `0` or `1`). Per rule 3 above, the default value of `0` must be omitted, so if a Boolean is included it must have a value of `1`.

While rule number 1. and 2. should be pretty straight forward and describe the default behavior of all protobuf encoders the author is aware of, the 3rd rule is more interesting. After a protobuf3 deserialization you cannot differentiate between unset fields and fields set to the default value<sup>3</sup>. At serialization level however, it is possible to set the fields with an empty value or omitting them entirely. This is a significant

difference to e.g. JSON where a property can be empty ( "", 0 ), null or undefined, leading to 3 different documents.

Omitting fields set to default values is valid because the parser must assign the default value to fields missing in the serialization<sup>4</sup>. For scalar types, omitting defaults is required by the spec<sup>5</sup>. For repeated fields, not serializing them is the only way to express empty lists. Enums must have a first element of numeric value 0, which is the default<sup>6</sup>. And message fields default to unset<sup>7</sup>.

Omitting defaults allows for some amount of forward compatibility: users of newer versions of a protobuf schema produce the same serialization as users of older versions as long as newly added fields are not used (i.e. set to their default value).

## Implementation

There are three main implementation strategies, ordered from the least to the most custom development:

- **Use a protobuf serializer that follows the above rules by default.** E.g. [gogoproto](#) is known to be compliant by in most cases, but not when certain annotations such as nullable = false are used. It might also be an option to configure an existing serializer accordingly.
- **Normalize default values before encoding them.** If your serializer follows rule 1. and 2. and allows you to explicitly unset fields for serialization, you can normalize default values to unset. This can be done when working with [protobuf.js](#):

```
const bytes = SignDoc.encode({
 bodyBytes: body.length > 0 ? body : null, // normalize empty bytes to unset
 authInfoBytes: authInfo.length > 0 ? authInfo : null, // normalize empty
 bytes to unset
 chainId: chainId || null, // normalize "" to unset
 accountNumber: accountNumber || null, // normalize 0 to unset
 accountSequence: accountSequence || null, // normalize 0 to unset
}).finish();
```

- **Use a hand-written serializer for the types you need.** If none of the above ways works for you, you can write a serializer yourself. For SignDoc this would look something like this in Go, building on existing protobuf utilities:

```
if !signDoc.body_bytes.empty() {
 buf.WriteUVarInt64(0xA) // wire type and field number for body_bytes
 buf.WriteUVarInt64(signDoc.body_bytes.length())
 buf.WriteBytes(signDoc.body_bytes)
}

if !signDoc.auth_info.empty() {
 buf.WriteUVarInt64(0x12) // wire type and field number for auth_info
 buf.WriteUVarInt64(signDoc.auth_info.length())
 buf.WriteBytes(signDoc.auth_info)
}

if !signDoc.chain_id.empty() {
 buf.WriteUVarInt64(0x1a) // wire type and field number for chain_id
 buf.WriteUVarInt64(signDoc.chain_id.length())
```

```

 buf.WriteBytes(signDoc.chain_id)
 }

 if signDoc.account_number != 0 {
 buf.WriteUVarInt64(0x20) // wire type and field number for account_number
 buf.WriteUVarInt(signDoc.account_number)
 }

 if signDoc.account_sequence != 0 {
 buf.WriteUVarInt64(0x28) // wire type and field number for
 account_sequence
 buf.WriteUVarInt(signDoc.account_sequence)
 }
}

```

## Test vectors

Given the protobuf definition Article.proto

```

package blog;
syntax = "proto3";

enum Type {
 UNSPECIFIED = 0;
 IMAGES = 1;
 NEWS = 2;
}

enum Review {
 UNSPECIFIED = 0;
 ACCEPTED = 1;
 REJECTED = 2;
}

message Article {
 string title = 1;
 string description = 2;
 uint64 created = 3;
 uint64 updated = 4;
 bool public = 5;
 bool promoted = 6;
 Type type = 7;
 Review review = 8;
 repeated string comments = 9;
 repeated string backlinks = 10;
}

```

serializing the values

```

title: "The world needs change 🌎"
description: ""
created: 1596806111080

```

```
updated: 0
public: true
promoted: false
type: Type.NEWS
review: Review.UNSPECIFIED
comments: ["Nice one", "Thank you"]
backlinks: []
```

must result in the serialization

```
0a1b54686520776f726c64206e65656473206368616e676520f09f8cb318e8bebec8bc2e280138024a084e0
```

When inspecting the serialized document, you see that every second field is omitted:

```
$ echo
0a1b54686520776f726c64206e65656473206368616e676520f09f8cb318e8bebec8bc2e280138024a084e0
| xxd -r -p | protoc --decode_raw
1: "The world needs change \360\237\214\263"
3: 1596806111080
5: 1
7: 2
9: "Nice one"
9: "Thank you"
```

## Consequences

Having such an encoding available allows us to get deterministic serialization for all protobuf documents we need in the context of Cosmos SDK signing.

### Positive

- Well defined rules that can be verified independent of a reference implementation
- Simple enough to keep the barrier to implement transaction signing low
- It allows us to continue to use 0 and other empty values in SignDoc, avoiding the need to work around 0 sequences. This does not imply the change from <https://github.com/cosmos/cosmos-sdk/pull/6949> should not be merged, but not too important anymore.

### Negative

- When implementing transaction signing, the encoding rules above must be understood and implemented.
- The need for rule number 3. adds some complexity to implementations.
- Some data structures may require custom code for serialization. Thus the code is not very portable
  - it will require additional work for each client implementing serialization to properly handle custom data structures.

### Neutral

### Usage in Cosmos SDK

For the reasons mentioned above ("Negative" section) we prefer to keep workarounds for shared data structure. Example: the aforementioned `TxRaw` is using raw bytes as a workaround. This allows them to

use any valid Protobuf library without the need of implementing a custom serializer that adheres to this standard (and related risks of bugs).

## References

- <sup>1</sup> When a message is serialized, there is no guaranteed order for how its known or unknown fields should be written. Serialization order is an implementation detail and the details of any particular implementation may change in the future. Therefore, protocol buffer parsers must be able to parse fields in any order. from <https://developers.google.com/protocol-buffers/docs/encoding#order>
- <sup>2</sup> [https://developers.google.com/protocol-buffers/docs/encoding#signed\\_integers](https://developers.google.com/protocol-buffers/docs/encoding#signed_integers)
- <sup>3</sup> Note that for scalar message fields, once a message is parsed there's no way of telling whether a field was explicitly set to the default value (for example whether a boolean was set to false) or just not set at all: you should bear this in mind when defining your message types. For example, don't have a boolean that switches on some behavior when set to false if you don't want that behavior to also happen by default. from <https://developers.google.com/protocol-buffers/docs/proto3#default>
- <sup>4</sup> When a message is parsed, if the encoded message does not contain a particular singular element, the corresponding field in the parsed object is set to the default value for that field. from <https://developers.google.com/protocol-buffers/docs/proto3#default>
- <sup>5</sup> Also note that if a scalar message field is set to its default, the value will not be serialized on the wire. from <https://developers.google.com/protocol-buffers/docs/proto3#default>
- <sup>6</sup> For enums, the default value is the first defined enum value, which must be 0. from <https://developers.google.com/protocol-buffers/docs/proto3#default>
- <sup>7</sup> For message fields, the field is not set. Its exact value is language-dependent. from <https://developers.google.com/protocol-buffers/docs/proto3#default>
- Encoding rules and parts of the reasoning taken from [canonical-proto3 Aaron Craelius](#)

# ADR 028: Public Key Addresses

## Changelog

- 2020/08/18: Initial version
- 2021/01/15: Analysis and algorithm update

## Status

Proposed

## Abstract

This ADR defines an address format for all addressable Cosmos SDK accounts. That includes: new public key algorithms, multisig public keys, and module accounts.

## Context

Issue [#3685](#) identified that public key address spaces are currently overlapping. We confirmed that it significantly decreases security of Cosmos SDK.

## Problem

An attacker can control an input for an address generation function. This leads to a birthday attack, which significantly decreases the security space. To overcome this, we need to separate the inputs for different kind of account types: a security break of one account type shouldn't impact the security of other account types.

## Initial proposals

One initial proposal was extending the address length and adding prefixes for different types of addresses.

@ethanfrey explained an alternate approach originally used in <https://github.com/iov-one/weave>:

*I spent quite a bit of time thinking about this issue while building weave... The other cosmos Sdk. Basically I define a condition to be a type and format as human readable string with some binary data appended. This condition is hashed into an Address (again at 20 bytes). The use of this prefix makes it impossible to find a preimage for a given address with a different condition (eg ed25519 vs secp256k1). This is explained in depth here <https://weave.readthedocs.io/en/latest/design/permissions.html> And the code is here, look mainly at the top where we process conditions. <https://github.com/iov-one/weave/blob/master/conditions.go>*

And explained how this approach should be sufficiently collision resistant:

*Yeah, AFAIK, 20 bytes should be collision resistance when the preimages are unique and not malleable. A space of  $2^{160}$  would expect some collision to be likely around  $2^{80}$  elements (birthday paradox). And if you want to find a collision for some existing element in the database, it is still  $2^{160}$ .  $2^{80}$  only is if all these elements are written to state. The good example you brought up was eg. a public key bytes being a valid public key on two algorithms supported by the codec. Meaning if either was broken, you would break accounts even if they were secured with the safer variant. This is only as the issue when no differentiating type info is present in the preimage (before hashing into an address). I would like to hear an argument if the 20 bytes space is an actual issue for security, as I would be happy to increase my address sizes in weave. I just figured cosmos and ethereum and bitcoin all use 20 bytes, it should be good enough. And the arguments above which made me feel it was secure. But I have not done a deeper analysis.*

This led to the first proposal (which we proved to be not good enough): we concatenate a key type with a public key, hash it and take the first 20 bytes of that hash, summarized as `sha256(keyTypePrefix || keybytes)[:20]`.

## Review and Discussions

In [#5694](#) we discussed various solutions. We agreed that 20 bytes it's not future proof, and extending the address length is the only way to allow addresses of different types, various signature types, etc. This disqualifies the initial proposal.

In the issue we discussed various modifications:

- Choice of the hash function.
- Move the prefix out of the hash function: `keyTypePrefix + sha256(keybytes)[:20]` [post-hash-prefix-proposal].
- Use double hashing: `sha256(keyTypePrefix + sha256(keybytes)[:20])` .
- Increase to keybytes hash slice from 20 byte to 32 or 40 bytes. We concluded that 32 bytes, produced by a good hash functions is future secure.

## Requirements

- Support currently used tools - we don't want to break an ecosystem, or add a long adaptation period. Ref: <https://github.com/cosmos/cosmos-sdk/issues/8041>
- Try to keep the address length small - addresses are widely used in state, both as part of a key and object value.

## Scope

This ADR only defines a process for the generation of address bytes. For end-user interactions with addresses (through the API, or CLI, etc.), we still use bech32 to format these addresses as strings. This ADR doesn't change that. Using Bech32 for string encoding gives us support for checksum error codes and handling of user typos.

## Decision

We define the following account types, for which we define the address function:

1. simple accounts: represented by a regular public key (ie: secp256k1, sr25519)
2. naive multisig: accounts composed by other addressable objects (ie: naive multisig)
3. composed accounts with a native address key (ie: bls, group module accounts)
4. module accounts: basically any accounts which cannot sign transactions and which are managed internally by modules

### Legacy Public Key Addresses Don't Change

Currently (Jan 2021), the only officially supported Cosmos SDK user accounts are `secp256k1` basic accounts and legacy amino multisig. They are used in existing Cosmos SDK zones. They use the following address formats:

- `secp256k1: ripemd160(sha256(pk_bytes))[:20]`
- legacy amino multisig: `sha256(aminoCdc.Marshal(pk))[:20]`

We don't want to change existing addresses. So the addresses for these two key types will remain the same.

The current multisig public keys use amino serialization to generate the address. We will retain those public keys and their address formatting, and call them "legacy amino" multisig public keys in protobuf. We will also create multisig public keys without amino addresses to be described below.

## Hash Function Choice

As in other parts of the Cosmos SDK, we will use `sha256`.

## Basic Address

We start with defining a base algorithm for generating addresses which we will call `Hash`. Notably, it's used for accounts represented by a single key pair. For each public key schema we have to have an associated `typ` string, explained in the next section. `hash` is the cryptographic hash function defined in the previous section.

```
const A_LEN = 32

func Hash(typ string, key []byte) []byte {
 return hash(hash(typ) + key)[:A_LEN]
}
```

The `+` is bytes concatenation, which doesn't use any separator.

This algorithm is the outcome of a consultation session with a professional cryptographer. Motivation: this algorithm keeps the address relatively small (length of the `typ` doesn't impact the length of the final address) and it's more secure than [post-hash-prefix-proposal] (which uses the first 20 bytes of a pubkey hash, significantly reducing the address space). Moreover the cryptographer motivated the choice of adding `typ` in the hash to protect against a switch table attack.

`address.Hash` is a low level function to generate *base* addresses for new key types. Example:

- BLS: `address.Hash("bls", pubkey)`

## Composed Addresses

For simple composed accounts (like a new naive multisig) we generalize the `address.Hash`. The address is constructed by recursively creating addresses for the sub accounts, sorting the addresses and composing them into a single address. It ensures that the ordering of keys doesn't impact the resulting address.

```
// We don't need a PubKey interface - we need anything which is addressable.
type Addressable interface {
 Address() []byte
}

func Composed(typ string, subaccounts []Addressable) []byte {
 addresses = map(subaccounts, \a -> LengthPrefix(a.Address()))
 addresses = sort(addresses)
 return address.Hash(typ, addresses[0] + ... + addresses[n])
}
```

The `typ` parameter should be a schema descriptor, containing all significant attributes with deterministic serialization (eg: utf8 string). `LengthPrefix` is a function which prepends 1 byte to the address. The value of that byte is the length of the address bits before prepending. The address must be at most 255 bits long. We are using `LengthPrefix` to eliminate conflicts - it assures, that for 2 lists of addresses: `as = {a1, a2, ..., an}` and `bs = {b1, b2, ..., bm}` such that every `bi` and `ai` is at most 255 long, `concatenate(map(as, (a) => LengthPrefix(a))) = map(bs, (b) => LengthPrefix(b))` if `as = bs`.

Implementation Tip: account implementations should cache addresses.

## Multisig Addresses

For a new multisig public keys, we define the `typ` parameter not based on any encoding scheme (amino or protobuf). This avoids issues with non-determinism in the encoding scheme.

Example:

```
package cosmos.crypto.multisig;

message PubKey {
 uint32 threshold = 1;
```

```

repeated google.protobuf.Any pubkeys = 2;
}

func (multisig PubKey) Address() {
 // first gather all nested pub keys
 var keys []address.Addressable // cryptotypes.PubKey implements Addressable
 for _, key := range multisig.Pubkeys {
 keys = append(keys, key.GetCachedValue().(cryptotypes.PubKey))
 }

 // form the type from the message name (cosmos.crypto.multisig.PubKey) and the
 threshold joined together
 prefix := fmt.Sprintf("%s/%d", proto.MessageName(multisig), multisig.Threshold)

 // use the Composed function defined above
 return address.Composed(prefix, keys)
}

```

## Derived Addresses

We must be able to cryptographically derive one address from another one. The derivation process must guarantee hash properties, hence we use the already defined `Hash` function:

```

func Derive(address, derivationKey []byte) []byte {
 return Hash(address, derivationKey)
}

```

## Module Account Addresses

A module account will have "module" type. Module accounts can have sub accounts. The submodule account will be created based on module name, and sequence of derivation keys. Typically, the first derivation key should be a class of the derived accounts. The derivation process has a defined order: module name, submodule key, subsubmodule key... An example module account is created using:

```
address.Module(moduleName, key)
```

An example sub-module account is created using:

```

groupPolicyAddresses := []byte{1}
address.Module(moduleName, groupPolicyAddresses, policyID)

```

The `address.Module` function is using `address.Hash` with "module" as the type argument, and byte representation of the module name concatenated with submodule key. The two last component must be uniquely separated to avoid potential clashes (example: modulename="ab" & submodulekey="bc" will have the same derivation key as modulename="a" & submodulekey="bbc"). We use a null byte ( '\x00' ) to separate module name from the submodule key. This works, because null byte is not a part of a valid module name. Finally, the sub-submodule accounts are created by applying the `Derive` function recursively. We could use `Derive` function also in the first step (rather than concatenating module name with zero byte

and the submodule key). We decided to do concatenation to avoid one level of derivation and speed up computation.

For backward compatibility with the existing `authtypes.NewModuleAddress`, we add a special case in `Module` function: when no derivation key is provided, we fallback to the "legacy" implementation.

```
func Module(moduleName string, derivationKeys ...[]byte) []byte{
 if len(derivationKeys) == 0 {
 return authtypes.NewModuleAddress(modulenName) // legacy case
 }
 submoduleAddress := Hash("module", []byte(moduleName) + 0 + key)
 return fold((a, k) => Derive(a, k), subsubKeys, submoduleAddress)
}
```

**Example 1** A lending BTC pool address would be:

```
btcPool := address.Module("lending", btc.Address())
```

If we want to create an address for a module account depending on more than one key, we can concatenate them:

```
btcAtomAMM := address.Module("amm", btc.Address() + atom.Address())
```

**Example 2** a smart-contract address could be constructed by:

```
smartContractAddr = Module("mySmartContractVM", smartContractsNamespace,
smartContractKey)

// which equals to:
smartContractAddr = Derived(
 Module("mySmartContractVM", smartContractsNamespace),
 []{smartContractKey})
```

## Schema Types

A `typ` parameter used in `Hash` function SHOULD be unique for each account type. Since all Cosmos SDK account types are serialized in the state, we propose to use the protobuf message name string.

Example: all public key types have a unique protobuf message type similar to:

```
package cosmos.crypto.sr25519;

message PubKey {
 bytes key = 1;
}
```

All protobuf messages have unique fully qualified names, in this example

`cosmos.crypto.sr25519.PubKey`. These names are derived directly from .proto files in a standardized

way and used in other places such as the type URL in `Any`s. We can easily obtain the name using `proto.MessageName(msg)`.

## Consequences

### Backwards Compatibility

This ADR is compatible with what was committed and directly supported in the Cosmos SDK repository.

### Positive

- a simple algorithm for generating addresses for new public keys, complex accounts and modules
- the algorithm generalizes *native composed keys*
- increased security and collision resistance of addresses
- the approach is extensible for future use-cases - one can use other address types, as long as they don't conflict with the address length specified here (20 or 32 bytes).
- support new account types.

### Negative

- addresses do not communicate key type, a prefixed approach would have done this
- addresses are 60% longer and will consume more storage space
- requires a refactor of KVStore store keys to handle variable length addresses

### Neutral

- protobuf message names are used as key type prefixes

## Further Discussions

Some accounts can have a fixed name or may be constructed in other way (eg: modules). We were discussing an idea of an account with a predefined name (eg: `me.regen`), which could be used by institutions. Without going into details, these kinds of addresses are compatible with the hash based addresses described here as long as they don't have the same length. More specifically, any special account address must not have a length equal to 20 or 32 bytes.

## Appendix: Consulting session

End of Dec 2020 we had a session with [Alan Szepieniec](#) to consult the approach presented above.

Alan general observations:

- we don't need 2-preimage resistance
- we need 32bytes address space for collision resistance
- when an attacker can control an input for object with an address then we have a problem with birthday attack
- there is an issue with smart-contracts for hashing
- sha2 mining can be used to breaking address pre-image

Hashing algorithm

- any attack breaking blake3 will break blake2
- Alan is pretty confident about the current security analysis of the blake hash algorithm. It was a finalist, and the author is well known in security analysis.

Algorithm:

- Alan recommends to hash the prefix: `address(pub_key) = hash(hash(key_type) + pub_key)[0:32]`, main benefits:
  - we are free to user arbitrary long prefix names
  - we still don't risk collisions
  - switch tables
- discussion about penalization -> about adding prefix post hash
- Aaron asked about post hash prefixes (`address(pub_key) = key_type + hash(pub_key)`) and differences. Alan noted that this approach has longer address space and it's stronger.

Algorithm for complex / composed keys:

- merging tree like addresses with same algorithm are fine

Module addresses: Should module addresses have different size to differentiate it?

- we will need to set a pre-image prefix for module addresse to keep them in 32-byte space:  
`hash(hash('module') + module_key)`
- Aaron observation: we already need to deal with variable length (to not break secp256k1 keys).

Discussion about arithmetic hash function for ZKP

- Posseidon / Rescue
- Problem: much bigger risk because we don't know much techniques and history of crypto-analysis of arithmetic constructions. It's still a new ground and area of active research.

Post quantum signature size

- Alan suggestion: Falcon: speed / size ration - very good.
- Aaron - should we think about it? Alan: based on early extrapolation this thing will get able to break EC cryptography in 2050 . But that's a lot of uncertainty. But there is magic happening with recursions / linking / simulation and that can speedup the progress.

Other ideas

- Let's say we use same key and two different address algorithms for 2 different use cases. Is it still safe to use it? Alan: if we want to hide the public key (which is not our use case), then it's less secure but there are fixes.

## References

- [Notes](#)

# ADR 029: Fee Grant Module

## Changelog

- 2020/08/18: Initial Draft
- 2021/05/05: Removed height based expiration support and simplified naming.

## Status

Accepted

## Context

In order to make blockchain transactions, the signing account must possess a sufficient balance of the right denomination in order to pay fees. There are classes of transactions where needing to maintain a wallet with sufficient fees is a barrier to adoption.

For instance, when proper permissions are setup, someone may temporarily delegate the ability to vote on proposals to a "burner" account that is stored on a mobile phone with only minimal security.

Other use cases include workers tracking items in a supply chain or farmers submitting field data for analytics or compliance purposes.

For all of these use cases, UX would be significantly enhanced by obviating the need for these accounts to always maintain the appropriate fee balance. This is especially true if we wanted to achieve enterprise adoption for something like supply chain tracking.

While one solution would be to have a service that fills up these accounts automatically with the appropriate fees, a better UX would be provided by allowing these accounts to pull from a common fee pool account with proper spending limits. A single pool would reduce the churn of making lots of small "fill up" transactions and also more effectively leverages the resources of the organization setting up the pool.

## Decision

As a solution we propose a module, `*/feegrant` which allows one account, the "granter" to grant another account, the "grantee" an allowance to spend the granter's account balance for fees within certain well-defined limits.

Fee allowances are defined by the extensible `FeeAllowanceI` interface:

```
type FeeAllowanceI {
 // Accept can use fee payment requested as well as timestamp of the current block
 // to determine whether or not to process this. This is checked in
 // Keeper.UseGrantedFees and the return values should match how it is handled
 there.
 //
 // If it returns an error, the fee payment is rejected, otherwise it is accepted.
 // The FeeAllowance implementation is expected to update it's internal state
 // and will be saved again after an acceptance.
 //
 // If remove is true (regardless of the error), the FeeAllowance will be deleted
 from storage
 // (eg. when it is used up). (See call to RevokeFeeAllowance in
 Keeper.UseGrantedFees)
 Accept(ctx sdk.Context, fee sdk.Coins, msgs []sdk.Msg) (remove bool, err error)

 // ValidateBasic should evaluate this FeeAllowance for internal consistency.
 // Don't allow negative amounts, or negative periods for example.
 ValidateBasic() error
}
```

Two basic fee allowance types, `BasicAllowance` and `PeriodicAllowance` are defined to support known use cases:

```

// BasicAllowance implements FeeAllowanceI with a one-time grant of tokens
// that optionally expires. The delegatee can use up to SpendLimit to cover fees.
message BasicAllowance {
 // spend_limit specifies the maximum amount of tokens that can be spent
 // by this allowance and will be updated as tokens are spent. If it is
 // empty, there is no spend limit and any amount of coins can be spent.
 repeated cosmos_sdk.v1.Coin spend_limit = 1;

 // expiration specifies an optional time when this allowance expires
 google.protobuf.Timestamp expiration = 2;
}

// PeriodicAllowance extends FeeAllowanceI to allow for both a maximum cap,
// as well as a limit per time period.
message PeriodicAllowance {
 BasicAllowance basic = 1;

 // period specifies the time duration in which period_spend_limit coins can
 // be spent before that allowance is reset
 google.protobuf.Duration period = 2;

 // period_spend_limit specifies the maximum number of coins that can be spent
 // in the period
 repeated cosmos_sdk.v1.Coin period_spend_limit = 3;

 // period_can_spend is the number of coins left to be spent before the
 period_reset time
 repeated cosmos_sdk.v1.Coin period_can_spend = 4;

 // period_reset is the time at which this period resets and a new one begins,
 // it is calculated from the start time of the first transaction after the
 // last period ended
 google.protobuf.Timestamp period_reset = 5;
}

```

Allowances can be granted and revoked using `MsgGrantAllowance` and `MsgRevokeAllowance`:

```

// MsgGrantAllowance adds permission for Grantee to spend up to Allowance
// of fees from the account of Granter.
message MsgGrantAllowance {
 string granter = 1;
 string grantee = 2;
 google.protobuf.Any allowance = 3;
}

// MsgRevokeAllowance removes any existing FeeAllowance from Granter to Grantee.
message MsgRevokeAllowance {
 string granter = 1;
 string grantee = 2;
}

```

In order to use allowances in transactions, we add a new field `granter` to the transaction `Fee` type:

```
package cosmos.tx.v1beta1;

message Fee {
 repeated cosmos.base.v1beta1.Coin amount = 1;
 uint64 gas_limit = 2;
 string payer = 3;
 string granter = 4;
}
```

`granter` must either be left empty or must correspond to an account which has granted a fee allowance to fee payer (either the first signer or the value of the `payer` field).

A new `AnteDecorator` named `DeductGrantedFeeDecorator` will be created in order to process transactions with `fee_payer` set and correctly deduct fees based on fee allowances.

## Consequences

### Positive

- improved UX for use cases where it is cumbersome to maintain an account balance just for fees

### Negative

### Neutral

- a new field must be added to the transaction `Fee` message and a new `AnteDecorator` must be created to use it

## References

- Blog article describing initial work: <https://medium.com/regen-network/hacking-the-cosmos-cosmwasm-and-key-management-a08b9f561d1b>
- Initial public specification: <https://gist.github.com/aaronc/b60628017352df5983791cad30babe56>
- Original subkeys proposal from B-harvest which influenced this design:  
<https://github.com/cosmos/cosmos-sdk/issues/4480>

## ADR 030: Authorization Module

### Changelog

- 2019-11-06: Initial Draft
- 2020-10-12: Updated Draft
- 2020-11-13: Accepted
- 2020-05-06: proto API updates, use `sdk.Msg` instead of `sdk.ServiceMsg` (the latter concept was removed from Cosmos SDK)
- 2022-04-20: Updated the `SendAuthorization` proto docs to clarify the `SpendLimit` is a required field. (Generic authorization can be used with bank msg type url to create limit less bank authorization)

## Status

Accepted

## Abstract

This ADR defines the `x/authz` module which allows accounts to grant authorizations to perform actions on behalf of that account to other accounts.

## Context

The concrete use cases which motivated this module include:

- the desire to delegate the ability to vote on proposals to other accounts besides the account which one has delegated stake
- "sub-keys" functionality, as originally proposed in [#4480](#) which is a term used to describe the functionality provided by this module together with the `fee_grant` module from [ADR 029](#) and the [group module](#).

The "sub-keys" functionality roughly refers to the ability for one account to grant some subset of its capabilities to other accounts with possibly less robust, but easier to use security measures. For instance, a master account representing an organization could grant the ability to spend small amounts of the organization's funds to individual employee accounts. Or an individual (or group) with a multisig wallet could grant the ability to vote on proposals to any one of the member keys.

The current implementation is based on work done by the [Gaian's team at Hackatom Berlin 2019](#).

## Decision

We will create a module named `authz` which provides functionality for granting arbitrary privileges from one account (the *grantee*) to another account (the *granteee*). Authorizations must be granted for a particular `Msg` service methods one by one using an implementation of `Authorization` interface.

## Types

Authorizations determine exactly what privileges are granted. They are extensible and can be defined for any `Msg` service method even outside of the module where the `Msg` method is defined.

`Authorization`'s reference `Msg`'s using their TypeURL.

### Authorization

```
type Authorization interface {
 proto.Message

 // MsgTypeURL returns the fully-qualified Msg TypeURL (as described in ADR 020),
 // which will process and accept or reject a request.
 MsgTypeURL() string

 // Accept determines whether this grant permits the provided sdk.Msg to be
 // performed, and if
 // so provides an upgraded authorization instance.
```

```

Accept(ctx sdk.Context, msg sdk.Msg) (AcceptResponse, error)

// ValidateBasic does a simple validation check that
// doesn't require access to any other information.
ValidateBasic() error
}

// AcceptResponse instruments the controller of an authz message if the request is
accepted
// and if it should be updated or deleted.
type AcceptResponse struct {
 // If Accept=true, the controller can accept and authorization and handle the
update.
 Accept bool
 // If Delete=true, the controller must delete the authorization object and
release
 // storage resources.
 Delete bool
 // Controller, who is calling Authorization.Accept must check if `Updated != nil`.
 // If yes,
 // it must use the updated version and handle the update on the storage level.
 Updated Authorization
}

```

For example a `SendAuthorization` like this is defined for `MsgSend` that takes a `SpendLimit` and updates it down to zero:

```

type SendAuthorization struct {
 // SpendLimit specifies the maximum amount of tokens that can be spent
 // by this authorization and will be updated as tokens are spent. This field is
required. (Generic authorization
 // can be used with bank msg type url to create limit less bank authorization).
 SpendLimit sdk.Coins
}

func (a SendAuthorization) MsgTypeURL() string {
 return sdk.MsgTypeURL(&MsgSend{})
}

func (a SendAuthorization) Accept(ctx sdk.Context, msg sdk.Msg)
(authz.AcceptResponse, error) {
 mSend, ok := msg.(*MsgSend)
 if !ok {
 return authz.AcceptResponse{}, sdkerrors.ErrInvalidType.Wrap("type
mismatch")
 }
 limitLeft, isNegative := a.SpendLimit.SafeSub(mSend.Amount)
 if isNegative {
 return authz.AcceptResponse{},
 sdkerrors.ErrInsufficientFunds.Wrapf("requested amount is more than spend limit")
 }
}

```

```

if limitLeft.IsZero() {
 return authz.AcceptResponse{Accept: true, Delete: true}, nil
}

return authz.AcceptResponse{Accept: true, Delete: false, Updated:
&SendAuthorization{SpendLimit: limitLeft}}, nil
}

```

A different type of capability for `MsgSend` could be implemented using the `Authorization` interface with no need to change the underlying `bank` module.

#### Small notes on `AcceptResponse`

- The `AcceptResponse.Accept` field will be set to `true` if the authorization is accepted. However, if it is rejected, the function `Accept` will raise an error (without setting `AcceptResponse.Accept` to `false`).
- The `AcceptResponse.Updated` field will be set to a non-nil value only if there is a real change to the authorization. If authorization remains the same (as is, for instance, always the case for a [GenericAuthorization](#)), the field will be `nil`.

## Msg Service

```

service Msg {
 // Grant grants the provided authorization to the grantee on the grantor's
 // account with the provided expiration time.
 rpc Grant(MsgGrant) returns (MsgGrantResponse);

 // Exec attempts to execute the provided messages using
 // authorizations granted to the grantee. Each message should have only
 // one signer corresponding to the grantor of the authorization.
 rpc Exec(MsgExec) returns (MsgExecResponse);

 // Revoke revokes any authorization corresponding to the provided method name on
 // the
 // grantor's account that has been granted to the grantee.
 rpc Revoke(MsgRevoke) returns (MsgRevokeResponse);
}

// Grant gives permissions to execute
// the provided method with expiration time.
message Grant {
 google.protobuf.Any authorization = 1 [(cosmos_proto.accepts_interface) =
"cosmos.authz.v1beta1.Authorization"];
 google.protobuf.Timestamp expiration = 2 [(gogoproto.stdtime) = true,
(gogoproto.nullable) = false];
}

message MsgGrant {
 string granter = 1;
 string grantee = 2;
}

```

```

Grant grant = 3 [(gogoproto.nullable) = false];
}

message MsgExecResponse {
 cosmos.base.abci.v1beta1.Result result = 1;
}

message MsgExec {
 string grantee = 1;
 // Authorization Msg requests to execute. Each msg must implement Authorization
 interface
 repeated google.protobuf.Any msgs = 2 [(cosmos_proto.accepts_interface) =
"cosmos.base.v1beta1.Msg"];
}

```

## Router Middleware

The `authz Keeper` will expose a `DispatchActions` method which allows other modules to send `Msg`s to the router based on `Authorization` grants:

```

type Keeper interface {
 // DispatchActions routes the provided msgs to their respective handlers if the
 grantee was granted an authorization
 // to send those messages by the first (and only) signer of each msg.
 DispatchActions(ctx sdk.Context, grantee sdk.AccAddress, msgs []sdk.Msg)
 sdk.Result
}

```

## CLI

### `tx exec` Method

When a CLI user wants to run a transaction on behalf of another account using `MsgExec`, they can use the `exec` method. For instance `gaiacli tx gov vote 1 yes --from <grantee> --generate-only | gaiacli tx authz exec --send-as <granter> --from <grantee>` would send a transaction like this:

```

MsgExec {
 Grantee: mykey,
 Msgs: []sdk.Msg{
 MsgVote {
 ProposalID: 1,
 Voter: cosmos3thsdg983egh823
 Option: Yes
 }
 }
}

```

`tx grant <grantee> <authorization> --from <granter>`

This CLI command will send a `MsgGrant` transaction. `authorization` should be encoded as JSON on the CLI.

```
tx revoke <grantee> <method-name> --from <granter>
```

This CLI command will send a `MsgRevoke` transaction.

## Built-in Authorizations

`SendAuthorization`

```
// SendAuthorization allows the grantee to spend up to spend_limit coins from
// the granter's account.
message SendAuthorization {
 repeated cosmos.base.v1beta1.Coin spend_limit = 1;
}
```

`GenericAuthorization`

```
// GenericAuthorization gives the grantee unrestricted permissions to execute
// the provided method on behalf of the granter's account.
message GenericAuthorization {
 option (cosmos_proto.implements_interface) = "Authorization";

 // Msg, identified by it's type URL, to grant unrestricted permissions to execute
 string msg = 1;
}
```

## Consequences

### Positive

- Users will be able to authorize arbitrary actions on behalf of their accounts to other users, improving key management for many use cases
- The solution is more generic than previously considered approaches and the `Authorization` interface approach can be extended to cover other use cases by SDK users

### Negative

### Neutral

## References

- Initial Hackatom implementation: <https://github.com/cosmos-gaians/cosmos-sdk/tree/hackatom/x/delegation>
- Post-Hackatom spec: <https://gist.github.com/aaronc/b60628017352df5983791cad30babe56#delegation-module>
- B-Harvest subkeys spec: <https://github.com/cosmos/cosmos-sdk/issues/4480>

## ADR 031: Protobuf Msg Services

## Changelog

- 2020-10-05: Initial Draft
- 2021-04-21: Remove `ServiceMsg`s to follow Protobuf `Any`'s spec, see [#9063](#).

## Status

Accepted

## Abstract

We want to leverage protobuf `service` definitions for defining `Msg`s which will give us significant developer UX improvements in terms of the code that is generated and the fact that return types will now be well defined.

## Context

Currently `Msg` handlers in the Cosmos SDK do have return values that are placed in the `data` field of the response. These return values, however, are not specified anywhere except in the golang handler code.

In early conversations [it was proposed](#) that `Msg` return types be captured using a protobuf extension field, ex:

```
package cosmos.gov;

message MsgSubmitProposal
 option (cosmos_proto.msg_return) = "uint64";
 string delegator_address = 1;
 string validator_address = 2;
 repeated sdk.Coin amount = 3;
}
```

This was never adopted, however.

Having a well-specified return value for `Msg`s would improve client UX. For instance, in `x/gov`, `MsgSubmitProposal` returns the proposal ID as a big-endian `uint64`. This isn't really documented anywhere and clients would need to know the internals of the Cosmos SDK to parse that value and return it to users.

Also, there may be cases where we want to use these return values programatically. For instance, <https://github.com/cosmos/cosmos-sdk/issues/7093> proposes a method for doing inter-module Ocaps using the `Msg` router. A well-defined return type would improve the developer UX for this approach.

In addition, handler registration of `Msg` types tends to add a bit of boilerplate on top of keepers and is usually done through manual type switches. This isn't necessarily bad, but it does add overhead to creating modules.

## Decision

We decide to use protobuf `service` definitions for defining `Msg`s as well as the code generated by them as a replacement for `Msg` handlers.

Below we define how this will look for the `SubmitProposal` message from `x/gov` module. We start with a `Msg` service definition:

```
package cosmos.gov;

service Msg {
 rpc SubmitProposal(MsgSubmitProposal) returns (MsgSubmitProposalResponse);
}

// Note that for backwards compatibility this uses MsgSubmitProposal as the request
// type instead of the more canonical MsgSubmitProposalRequest
message MsgSubmitProposal {
 google.protobuf.Any content = 1;
 string proposer = 2;
}

message MsgSubmitProposalResponse {
 uint64 proposal_id;
}
```

While this is most commonly used for gRPC, overloading protobuf `service` definitions like this does not violate the intent of the [protobuf spec](#) which says:

*If you don't want to use gRPC, it's also possible to use protocol buffers with your own RPC implementation. With this approach, we would get an auto-generated `MsgServer` interface:*

In addition to clearly specifying return types, this has the benefit of generating client and server code. On the server side, this is almost like an automatically generated keeper method and could maybe be used instead of keepers eventually (see [#7093](#)):

```
package gov

type MsgServer interface {
 SubmitProposal(context.Context, *MsgSubmitProposal) (*MsgSubmitProposalResponse,
 error)
}
```

On the client side, developers could take advantage of this by creating RPC implementations that encapsulate transaction logic. Protobuf libraries that use asynchronous callbacks, like [protobuf.js](#) could use this to register callbacks for specific messages even for transactions that include multiple `Msg`s.

Each `Msg` service method should have exactly one request parameter: its corresponding `Msg` type. For example, the `Msg` service method `/cosmos.gov.v1beta1.Msg/SubmitProposal` above has exactly one request parameter, namely the `Msg` type `/cosmos.gov.v1beta1.MsgSubmitProposal`. It is important the reader understands clearly the nomenclature difference between a `Msg` service (a Protobuf service) and a `Msg` type (a Protobuf message), and the differences in their fully-qualified name.

This convention has been decided over the more canonical `Msg...Request` names mainly for backwards compatibility, but also for better readability in `TxBody.messages` (see [Encoding section](#) below):

```
transactions containing /cosmos.gov.MsgSubmitProposal read better than those containing
/cosmos.gov.v1beta1.MsgSubmitProposalRequest .
```

One consequence of this convention is that each `Msg` type can be the request parameter of only one `Msg` service method. However, we consider this limitation a good practice in explicitness.

## Encoding

Encoding of transactions generated with `Msg` services do not differ from current Protobuf transaction encoding as defined in [ADR-020](#). We are encoding `Msg` types (which are exactly `Msg` service methods' request parameters) as `Any` in `Tx`'s which involves packing the binary-encoded `Msg` with its type URL.

## Decoding

Since `Msg` types are packed into `Any`, decoding transactions messages are done by unpacking `Any`'s into `Msg` types. For more information, please refer to [ADR-020](#).

## Routing

We propose to add a `msg_service_router` in `BaseApp`. This router is a key/value map which maps `Msg` types' `type_url`'s to their corresponding `Msg` service method handler. Since there is a 1-to-1 mapping between `Msg` types and `Msg` service method, the `msg_service_router` has exactly one entry per `Msg` service method.

When a transaction is processed by `BaseApp` (in `CheckTx` or in `DeliverTx`), its `TxBody.messages` are decoded as `Msg`'s. Each `Msg`'s `type_url` is matched against an entry in the `msg_service_router`, and the respective `Msg` service method handler is called.

For backward compatibility, the old handlers are not removed yet. If `BaseApp` receives a legacy `Msg` with no corresponding entry in the `msg_service_router`, it will be routed via its legacy `Route()` method into the legacy handler.

## Module Configuration

In [ADR 021](#), we introduced a method `RegisterQueryService` to `AppModule` which allows for modules to register gRPC queriers.

To register `Msg` services, we attempt a more extensible approach by converting `RegisterQueryService` to a more generic `RegisterServices` method:

```
type AppModule interface {
 RegisterServices(Configurator)
 ...
}

type Configurator interface {
 QueryServer() grpc.Server
 MsgServer() grpc.Server
}

// example module:
func (am AppModule) RegisterServices(cfg Configurator) {
```

```
 types.RegisterQueryServer(cfg.QueryServer(), keeper)
 types.RegisterMsgServer(cfg.MsgServer(), keeper)
}
```

The `RegisterServices` method and the `Configurator` interface are intended to evolve to satisfy the use cases discussed in [#7093](#) and [#7122](#).

When `Msg` services are registered, the framework *should* verify that all `Msg` types implement the `sdk.Msg` interface and throw an error during initialization rather than later when transactions are processed.

## Msg Service Implementation

Just like query services, `Msg` service methods can retrieve the `sdk.Context` from the `context.Context` parameter method using the `sdk.UnwrapSDKContext` method:

```
package gov

func (k Keeper) SubmitProposal(goCtx context.Context, params
 *types.MsgSubmitProposal) (*MsgSubmitProposalResponse, error) {
 ctx := sdk.UnwrapSDKContext(goCtx)
 ...
}
```

The `sdk.Context` should have an `EventManager` already attached by BaseApp's `msg_service_router`.

Separate handler definition is no longer needed with this approach.

## Consequences

This design changes how a module functionality is exposed and accessed. It deprecates the existing `Handler` interface and `AppModule.Route` in favor of [Protocol Buffer Services](#) and Service Routing described above. This dramatically simplifies the code. We don't need to create handlers and keepers any more. Use of Protocol Buffer auto-generated clients clearly separates the communication interfaces between the module and a modules user. The control logic (aka handlers and keepers) is not exposed any more. A module interface can be seen as a black box accessible through a client API. It's worth to note that the client interfaces are also generated by Protocol Buffers.

This also allows us to change how we perform functional tests. Instead of mocking `AppModules` and `Router`, we will mock a client (server will stay hidden). More specifically: we will never mock `moduleA.MsgServer` in `moduleB`, but rather `moduleA.MsgClient`. One can think about it as working with external services (eg DBs, or online servers...). We assume that the transmission between clients and servers is correctly handled by generated Protocol Buffers.

Finally, closing a module to client API opens desirable OCAP patterns discussed in ADR-033. Since server implementation and interface is hidden, nobody can hold "keepers"/servers and will be forced to relay on the client interface, which will drive developers for correct encapsulation and software engineering patterns.

## Pros

- communicates return type clearly
- manual handler registration and return type marshaling is no longer needed, just implement the interface and register it
- communication interface is automatically generated, the developer can now focus only on the state transition methods - this would improve the UX of [#7093](#) approach (1) if we chose to adopt that
- generated client code could be useful for clients and tests
- dramatically reduces and simplifies the code

## Cons

- using `service` definitions outside the context of gRPC could be confusing (but doesn't violate the proto3 spec)

## References

- [Initial Github Issue #7122](#)
- [proto 3 Language Guide: Defining Services](#)
- [Initial pre- Any . Msg . designs](#)
- [ADR 020](#)
- [ADR 021](#)

# ADR 032: Typed Events

## Changelog

- 28-Sept-2020: Initial Draft

## Authors

- Anil Kumar (@anilcse)
- Jack Zampolin (@jackzampolin)
- Adam Bozanich (@boz)

## Status

Proposed

## Abstract

Currently in the Cosmos SDK, events are defined in the handlers for each message as well as `BeginBlock` and `EndBlock`. Each module doesn't have types defined for each event, they are implemented as `map[string]string`. Above all else this makes these events difficult to consume as it requires a great deal of raw string matching and parsing. This proposal focuses on updating the events to use **typed events** defined in each module such that emitting and subscribing to events will be much easier. This workflow comes from the experience of the Akash Network team.

## Context

Currently in the Cosmos SDK, events are defined in the handlers for each message, meaning each module doesn't have a canonical set of types for each event. Above all else this makes these events difficult to consume as it requires a great deal of raw string matching and parsing. This proposal focuses on updating

the events to use **typed events** defined in each module such that emitting and subscribing to events will be much easier. This workflow comes from the experience of the Akash Network team.

[Our platform](#) requires a number of programmatic on chain interactions both on the provider (datacenter - to bid on new orders and listen for leases created) and user (application developer - to send the app manifest to the provider) side. In addition the Akash team is now maintaining the IBC [relayer](#), another very event driven process. In working on these core pieces of infrastructure, and integrating lessons learned from Kubernetes development, our team has developed a standard method for defining and consuming typed events in Cosmos SDK modules. We have found that it is extremely useful in building this type of event driven application.

As the Cosmos SDK gets used more extensively for apps like `peggy`, other peg zones, IBC, DeFi, etc... there will be an exploding demand for event driven applications to support new features desired by users. We propose upstreaming our findings into the Cosmos SDK to enable all Cosmos SDK applications to quickly and easily build event driven apps to aid their core application. Wallets, exchanges, explorers, and defi protocols all stand to benefit from this work.

If this proposal is accepted, users will be able to build event driven Cosmos SDK apps in go by just writing `EventHandler`s for their specific event types and passing them to `EventEmitters` that are defined in the Cosmos SDK.

The end of this proposal contains a detailed example of how to consume events after this refactor.

This proposal is specifically about how to consume these events as a client of the blockchain, not for intermodule communication.

## Decision

**Step-1:** Implement additional functionality in the `types` package: `EmitTypedEvent` and `ParseTypedEvent` functions

```
// types/events.go

// EmitTypedEvent takes typed event and emits converting it into sdk.Event
func (em *EventManager) EmitTypedEvent(event proto.Message) error {
 evtType := proto.MessageName(event)
 evtJSON, err := codec.ProtoMarshalJSON(event)
 if err != nil {
 return err
 }

 var attrMap map[string]json.RawMessage
 err = json.Unmarshal(evtJSON, &attrMap)
 if err != nil {
 return err
 }

 attrs := []abci.EventAttribute{
 for k, v := range attrMap {
 attrs = append(attrs, abci.EventAttribute{
 Key: []byte(k),
 Value: v,
 })
 }
 }

 em.EmitEvent(sdk.NewEvent("cosmos.event", attrs))
}
```

```

 })
 }

 em.EmitEvent(Event{
 Type: evtType,
 Attributes: attrs,
 })

 return nil
}

// ParseTypedEvent converts abci.Event back to typed event
func ParseTypedEvent(event abci.Event) (proto.Message, error) {
 concreteGoType := proto.MessageType(event.Type)
 if concreteGoType == nil {
 return nil, fmt.Errorf("failed to retrieve the message of type %q",
event.Type)
 }

 var value reflect.Value
 if concreteGoType.Kind() == reflect.Ptr {
 value = reflect.New(concreteGoType.Elem())
 } else {
 value = reflect.Zero(concreteGoType)
 }

 protoMsg, ok := value.Interface().(proto.Message)
 if !ok {
 return nil, fmt.Errorf("%q does not implement proto.Message", event.Type)
 }

 attrMap := make(map[string]json.RawMessage)
 for _, attr := range event.Attributes {
 attrMap[string(attr.Key)] = attr.Value
 }

 attrBytes, err := json.Marshal(attrMap)
 if err != nil {
 return nil, err
 }

 err = jsonpb.Unmarshal(strings.NewReader(string(attrBytes)), protoMsg)
 if err != nil {
 return nil, err
 }

 return protoMsg, nil
}

```

Here, the `EmitTypedEvent` is a method on `EventManager` which takes typed event as input and apply json serialization on it. Then it maps the JSON key/value pairs to `event.Attributes` and emits it in form

of `sdk.Event` . `Event.Type` will be the type URL of the proto message.

When we subscribe to emitted events on the CometBFT websocket, they are emitted in the form of an `abci.Event` . `ParseTypedEvent` parses the event back to its original proto message.

**Step-2:** Add proto definitions for typed events for msgs in each module:

For example, let's take `MsgSubmitProposal` of `gov` module and implement this event's type.

```
// proto/cosmos/gov/v1beta1/gov.proto
// Add typed event definition

package cosmos.gov.v1beta1;

message EventSubmitProposal {
 string from_address = 1;
 uint64 proposal_id = 2;
 TextProposal proposal = 3;
}
```

**Step-3:** Refactor event emission to use the typed event created and emit using `sdk.EmitTypedEvent` :

```
// x/gov/handler.go
func handleMsgSubmitProposal(ctx sdk.Context, keeper keeper.Keeper, msg
types.MsgSubmitProposalI) (*sdk.Result, error) {
 ...
 types.ContextEventManager().EmitTypedEvent(
 &EventSubmitProposal{
 FromAddress: fromAddress,
 ProposalId: id,
 Proposal: proposal,
 },
)
 ...
}
```

## How to subscribe to these typed events in Client

*NOTE: Full code example below*

Users will be able to subscribe using `client.Context.Client.Subscribe` and consume events which are emitted using `EventHandler`s.

Akash Network has built a simple `pubsub` . This can be used to subscribe to `abci.Events` and `publish` them as typed events.

Please see the below code sample for more detail on this flow looks for clients.

## Consequences

### Positive

- Improves consistency of implementation for the events currently in the Cosmos SDK
- Provides a much more ergonomic way to handle events and facilitates writing event driven applications
- This implementation will support a middleware ecosystem of `EventHandler`s

## Negative

## Detailed code example of publishing events

This ADR also proposes adding affordances to emit and consume these events. This way developers will only need to write `EventHandler`s which define the actions they desire to take.

```
// EventEmitter is a type that describes event emitter functions
// This should be defined in `types/events.go`
type EventEmitter func(context.Context, client.Context, ...EventHandler) error

// EventHandler is a type of function that handles events coming out of the event
bus
// This should be defined in `types/events.go`
type EventHandler func(proto.Message) error

// Sample use of the functions below
func main() {
 ctx, cancel := context.WithCancel(context.Background())

 if err := TxEmitter(ctx, client.Context{}.WithNodeURI("tcp://localhost:26657"),
SubmitProposalEventHandler); err != nil {
 cancel()
 panic(err)
 }

 return
}

// SubmitProposalEventHandler is an example of an event handler that prints proposal
details
// when any EventSubmitProposal is emitted.
func SubmitProposalEventHandler(ev proto.Message) (err error) {
 switch event := ev.(type) {
 // Handle governance proposal events creation events
 case govtypes.EventSubmitProposal:
 // Users define business logic here e.g.
 fmt.Println(ev.FromAddress, ev.ProposalId, ev.Proposal)
 return nil
 default:
 return nil
 }
}

// TxEmitter is an example of an event emitter that emits just transaction events.
This can and
// should be implemented somewhere in the Cosmos SDK. The Cosmos SDK can include an
```

```

EventEmitters for tm.event='Tx'
// and/or tm.event='NewBlock' (the new block events may contain typed events)
func TxEmitter(ctx context.Context, cliCtx client.Context, ehs ...EventHandler) (err
error) {
 // Instantiate and start CometBFT RPC client
 client, err := cliCtx.GetNode()
 if err != nil {
 return err
 }

 if err = client.Start(); err != nil {
 return err
 }

 // Start the pubsub bus
 bus := pubsub.NewBus()
 defer bus.Close()

 // Initialize a new error group
 eg, ctx := errgroup.WithContext(ctx)

 // Publish chain events to the pubsub bus
 eg.Go(func() error {
 return PublishChainTxEvents(ctx, client, bus, simapp.ModuleBasics)
 })

 // Subscribe to the bus events
 subscriber, err := bus.Subscribe()
 if err != nil {
 return err
 }

 // Handle all the events coming out of the bus
 eg.Go(func() error {
 var err error
 for {
 select {
 case <-ctx.Done():
 return nil
 case <-subscriber.Done():
 return nil
 case ev := <-subscriber.Events():
 for _, eh := range ehs {
 if err = eh(ev); err != nil {
 break
 }
 }
 }
 }
 return nil
 })
}

```

```

 return group.Wait()
 }

// PublishChainTxEvents events using cmtclient. Waits on context shutdown signals to
exit.
func PublishChainTxEvents(ctx context.Context, client cmtclient.EventsClient, bus
pubsub.Bus, mb module.BasicManager) (err error) {
 // Subscribe to transaction events
 txch, err := client.Subscribe(ctx, "txevents", "tm.event='Tx'", 100)
 if err != nil {
 return err
 }

 // Unsubscribe from transaction events on function exit
 defer func() {
 err = client.UnsubscribeAll(ctx, "txevents")
 }()
}

// Use errgroup to manage concurrency
g, ctx := errgroup.WithContext(ctx)

// Publish transaction events in a goroutine
g.Go(func() error {
 var err error
 for {
 select {
 case <-ctx.Done():
 break
 case ed := <-ch:
 switch evt := ed.Data.(type) {
 case cmttypes.EventDataTx:
 if !evt.Result.IsOK() {
 continue
 }
 // range over events, parse them using the basic manager and
 // send them to the pubsub bus
 for _, abciEv := range events {
 typedEvent, err := sdk.ParseTypedEvent(abciEv)
 if err != nil {
 return err
 }
 if err := bus.Publish(typedEvent); err != nil {
 bus.Close()
 return
 }
 continue
 }
 }
 }
 }
 return err
}))
```

```
// Exit on error or context cancelation
return g.Wait()
}
```

## References

- [Publish Custom Events via a bus](#)
- [Consuming the events in Client](#)

# ADR 033: Protobuf-based Inter-Module Communication

## Changelog

- 2020-10-05: Initial Draft

## Status

Proposed

## Abstract

This ADR introduces a system for permissioned inter-module communication leveraging the protobuf `Query` and `Msg` service definitions defined in [ADR 021](#) and [ADR 031](#) which provides:

- stable protobuf based module interfaces to potentially later replace the keeper paradigm
- stronger inter-module object capabilities (OCAPs) guarantees
- module accounts and sub-account authorization

## Context

In the current Cosmos SDK documentation on the [Object-Capability Model](#), it is stated that:

*We assume that a thriving ecosystem of Cosmos SDK modules that are easy to compose into a blockchain application will contain faulty or malicious modules.*

There is currently not a thriving ecosystem of Cosmos SDK modules. We hypothesize that this is in part due to:

1. lack of a stable v1.0 Cosmos SDK to build modules off of. Module interfaces are changing, sometimes dramatically, from point release to point release, often for good reasons, but this does not create a stable foundation to build on.
2. lack of a properly implemented object capability or even object-oriented encapsulation system which makes refactors of module keeper interfaces inevitable because the current interfaces are poorly constrained.

## x/bank Case Study

Currently the `x/bank` keeper gives pretty much unrestricted access to any module which references it. For instance, the `SetBalance` method allows the caller to set the balance of any account to anything,

bypassing even proper tracking of supply.

There appears to have been some later attempts to implement some semblance of OCAPs using module-level minting, staking and burning permissions. These permissions allow a module to mint, burn or delegate tokens with reference to the module's own account. These permissions are actually stored as a `[]string` array on the `ModuleAccount` type in state.

However, these permissions don't really do much. They control what modules can be referenced in the `MintCoins`, `BurnCoins` and `DelegateCoins***` methods, but for one there is no unique object capability token that controls access — just a simple string. So the `x/upgrade` module could mint tokens for the `x/staking` module simple by calling `MintCoins("staking")`. Furthermore, all modules which have access to these keeper methods, also have access to `SetBalance` negating any other attempt at OCAPs and breaking even basic object-oriented encapsulation.

## Decision

Based on [ADR-021](#) and [ADR-031](#), we introduce the Inter-Module Communication framework for secure module authorization and OCAPs. When implemented, this could also serve as an alternative to the existing paradigm of passing keepers between modules. The approach outlined here-in is intended to form the basis of a Cosmos SDK v1.0 that provides the necessary stability and encapsulation guarantees that allow a thriving module ecosystem to emerge.

Of particular note — the decision is to *enable* this functionality for modules to adopt at their own discretion. Proposals to migrate existing modules to this new paradigm will have to be a separate conversation, potentially addressed as amendments to this ADR.

### New "Keeper" Paradigm

In [ADR 021](#), a mechanism for using protobuf service definitions to define queriers was introduced and in [ADR 31](#), a mechanism for using protobuf service to define `Msg`s was added. Protobuf service definitions generate two golang interfaces representing the client and server sides of a service plus some helper code. Here is a minimal example for the bank `cosmos.bank.Msg/Send` message type:

```
package bank

type MsgClient interface {
 Send(context.Context, *MsgSend, opts ...grpc.CallOption) (*MsgSendResponse,
 error)
}

type MsgServer interface {
 Send(context.Context, *MsgSend) (*MsgSendResponse, error)
}
```

[ADR 021](#) and [ADR 31](#) specifies how modules can implement the generated `QueryServer` and `MsgServer` interfaces as replacements for the legacy queriers and `Msg` handlers respectively.

In this ADR we explain how modules can make queries and send `Msg`s to other modules using the generated `QueryClient` and `MsgClient` interfaces and propose this mechanism as a replacement for the existing `Keeper` paradigm. To be clear, this ADR does not necessitate the creation of new protobuf

definitions or services. Rather, it leverages the same proto based service interfaces already used by clients for inter-module communication.

Using this `QueryClient` / `MsgClient` approach has the following key benefits over exposing keepers to external modules:

1. Protobuf types are checked for breaking changes using `buf` and because of the way protobuf is designed this will give us strong backwards compatibility guarantees while allowing for forward evolution.
2. The separation between the client and server interfaces will allow us to insert permission checking code in between the two which checks if one module is authorized to send the specified `Msg` to the other module providing a proper object capability system (see below).
3. The router for inter-module communication gives us a convenient place to handle rollback of transactions, enabling atomicity of operations ([currently a problem](#)). Any failure within a module-to-module call would result in a failure of the entire transaction

This mechanism has the added benefits of:

- reducing boilerplate through code generation, and
- allowing for modules in other languages either via a VM like CosmWasm or sub-processes using gRPC

## Inter-module Communication

To use the `Client` generated by the protobuf compiler we need a `grpc.ClientConn` [interface](#) implementation. For this we introduce a new type, `ModuleKey`, which implements the `grpc.ClientConn` interface. `ModuleKey` can be thought of as the "private key" corresponding to a module account, where authentication is provided through use of a special `Invoker()` function, described in more detail below.

Blockchain users (external clients) use their account's private key to sign transactions containing `Msg`'s where they are listed as signers (each message specifies required signers with `Msg.GetSigner`). The authentication checks is performed by `AnteHandler`.

Here, we extend this process, by allowing modules to be identified in `Msg.GetSigners`. When a module wants to trigger the execution a `Msg` in another module, its `ModuleKey` acts as the sender (through the `ClientConn` interface we describe below) and is set as a sole "signer". It's worth to note that we don't use any cryptographic signature in this case. For example, module `A` could use its `A.ModuleKey` to create `MsgSend` object for `/cosmos.bank.Msg/Send` transaction. `MsgSend` validation will assure that the `from` account (`A.ModuleKey` in this case) is the signer.

Here's an example of a hypothetical module `foo` interacting with `x/bank`:

```
package foo

type FooMsgServer {
 // ...

 bankQuery bank.QueryClient
 bankMsg bank.MsgClient
}

func NewFooMsgServer(moduleKey RootModuleKey, ...) FooMsgServer {
```

```

// ...

return FooMsgServer {
 // ...
 moduleKey: moduleKey,
 bankQuery: bank.NewQueryClient(moduleKey),
 bankMsg: bank.NewMsgClient(moduleKey),
}
}

func (foo *FooMsgServer) Bar(ctx context.Context, req *MsgBarRequest)
(*MsgBarResponse, error) {
 balance, err := foo.bankQuery.Balance(&bank.QueryBalanceRequest{Address:
fooMsgServer.moduleKey.Address(), Denom: "foo"})

 ...

 res, err := foo.bankMsg.Send(ctx, &bank.MsgSendRequest{FromAddress:
fooMsgServer.moduleKey.Address(), ...})

 ...
}

```

This design is also intended to be extensible to cover use cases of more fine grained permissioning like minting by denom prefix being restricted to certain modules (as discussed in [#7459](#)).

### **ModuleKey S and ModuleID S**

A `ModuleKey` can be thought of as a "private key" for a module account and a `ModuleID` can be thought of as the corresponding "public key". From the [ADR 028](#), modules can have both a root module account and any number of sub-accounts or derived accounts that can be used for different pools (ex. staking pools) or managed accounts (ex. group accounts). We can also think of module sub-accounts as similar to derived keys - there is a root key and then some derivation path. `ModuleID` is a simple struct which contains the module name and optional "derivation" path, and forms its address based on the `AddressHash` method from [the ADR-028](#):

```

type ModuleID struct {
 ModuleName string
 Path []byte
}

func (key ModuleID) Address() []byte {
 return AddressHash(key.ModuleName, key.Path)
}

```

In addition to being able to generate a `ModuleID` and address, a `ModuleKey` contains a special function called `Invoker` which is the key to safe inter-module access. The `Invoker` creates an `InvokeFn` closure which is used as an `Invoke` method in the `grpc.ClientConn` interface and under the hood is able to route messages to the appropriate `Msg` and `Query` handlers performing appropriate security checks on `Msg`s. This allows for even safer inter-module access than keeper's whose private member

variables could be manipulated through reflection. Golang does not support reflection on a function closure's captured variables and direct manipulation of memory would be needed for a truly malicious module to bypass the `ModuleKey` security.

The two `ModuleKey` types are `RootModuleKey` and `DerivedModuleKey` :

```
type Invoker func(callInfo CallInfo) func(ctx context.Context, request, response
interface{}, opts ...interface{}) error

type CallInfo {
 Method string
 Caller ModuleID
}

type RootModuleKey struct {
 moduleName string
 invoker Invoker
}

func (rm RootModuleKey) Derive(path []byte) DerivedModuleKey { /* ... */}

type DerivedModuleKey struct {
 moduleName string
 path []byte
 invoker Invoker
}
```

A module can get access to a `DerivedModuleKey`, using the `Derive(path []byte)` method on `RootModuleKey` and then would use this key to authenticate `Msg`s from a sub-account. Ex:

```
package foo

func (fooMsgServer *MsgServer) Bar(ctx context.Context, req *MsgBar)
(*MsgBarResponse, error) {
 derivedKey := fooMsgServer.moduleKey.Derive(req.SomePath)
 bankMsgClient := bank.NewMsgClient(derivedKey)
 res, err := bankMsgClient.Balance(ctx, &bank.MsgSend{FromAddress:
derivedKey.Address(), ...})
 ...
}
```

In this way, a module can gain permissioned access to a root account and any number of sub-accounts and send authenticated `Msg`s from these accounts. The `Invoker callInfo.Caller` parameter is used under the hood to distinguish between different module accounts, but either way the function returned by `Invoker` only allows `Msg`s from either the root or a derived module account to pass through.

Note that `Invoker` itself returns a function closure based on the `CallInfo` passed in. This will allow client implementations in the future that cache the invoke function for each method type avoiding the overhead of hash table lookup. This would reduce the performance overhead of this inter-module communication method to the bare minimum required for checking permissions.

To re-iterate, the closure only allows access to authorized calls. There is no access to anything else regardless of any name impersonation.

Below is a rough sketch of the implementation of `grpc.ClientConn.Invoke` for `RootModuleKey`:

```
func (key RootModuleKey) Invoke(ctx context.Context, method string, args, reply
interface{}, opts ...grpc.CallOption) error {
 f := key.invoker(CallInfo {Method: method, Caller: ModuleID {ModuleName:
key.moduleName}}})
 return f(ctx, args, reply)
}
```

## AppModule Wiring and Requirements

In [ADR 031](#), the `AppModule.RegisterService(Configurator)` method was introduced. To support inter-module communication, we extend the `Configurator` interface to pass in the `ModuleKey` and to allow modules to specify their dependencies on other modules using `RequireServer()`:

```
type Configurator interface {
 MsgServer() grpc.Server
 QueryServer() grpc.Server

 ModuleKey() ModuleKey
 RequireServer(msgServer interface{})
}
```

The `ModuleKey` is passed to modules in the `RegisterService` method itself so that `RegisterServices` serves as a single entry point for configuring module services. This is intended to also have the side-effect of greatly reducing boilerplate in `app.go`. For now, `ModuleKey`s will be created based on `AppModuleBasic.Name()`, but a more flexible system may be introduced in the future. The `ModuleManager` will handle creation of module accounts behind the scenes.

Because modules do not get direct access to each other anymore, modules may have unfulfilled dependencies. To make sure that module dependencies are resolved at startup, the `Configurator.RequireServer` method should be added. The `ModuleManager` will make sure that all dependencies declared with `RequireServer` can be resolved before the app starts. An example module `foo` could declare it's dependency on `x/bank` like this:

```
package foo

func (am AppModule) RegisterServices(cfg Configurator) {
 cfg.RequireServer((*bank.QueryServer)(nil))
 cfg.RequireServer((*bank.MsgServer)(nil))
}
```

## Security Considerations

In addition to checking for `ModuleKey` permissions, a few additional security precautions will need to be taken by the underlying router infrastructure.

## Recursion and Re-entry

Recursive or re-entrant method invocations pose a potential security threat. This can be a problem if Module A calls Module B and Module B calls module A again in the same call.

One basic way for the router system to deal with this is to maintain a call stack which prevents a module from being referenced more than once in the call stack so that there is no re-entry. A `map[string]interface{}` table in the router could be used to perform this security check.

## Queries

Queries in Cosmos SDK are generally un-permissioned so allowing one module to query another module should not pose any major security threats assuming basic precautions are taken. The basic precaution that the router system will need to take is making sure that the `sdk.Context` passed to query methods does not allow writing to the store. This can be done for now with a `CacheMultiStore` as is currently done for `BaseApp` queries.

## Internal Methods

In many cases, we may wish for modules to call methods on other modules which are not exposed to clients at all. For this purpose, we add the `InternalServer` method to `Configurator`:

```
type Configurator interface {
 MsgServer() grpc.Server
 QueryServer() grpc.Server
 InternalServer() grpc.Server
}
```

As an example, `x/slashing`'s `Slash` must call `x/staking`'s `Slash`, but we don't want to expose `x/staking`'s `Slash` to end users and clients.

Internal protobuf services will be defined in a corresponding `internal.proto` file in the given module's proto package.

Services registered against `InternalServer` will be callable from other modules but not by external clients.

An alternative solution to internal-only methods could involve hooks / plugins as discussed [here](#). A more detailed evaluation of a hooks / plugin system will be addressed later in follow-ups to this ADR or as a separate ADR.

## Authorization

By default, the inter-module router requires that messages are sent by the first signer returned by `GetSigners`. The inter-module router should also accept authorization middleware such as that provided by [ADR 030](#). This middleware will allow accounts to otherwise specific module accounts to perform actions on their behalf. Authorization middleware should take into account the need to grant certain modules effectively "admin" privileges to other modules. This will be addressed in separate ADRs or updates to this ADR.

## Future Work

Other future improvements may include:

- custom code generation that:
  - simplifies interfaces (ex. generates code with `sdk.Context` instead of `context.Context`)
  - optimizes inter-module calls - for instance caching resolved methods after first invocation
- combining `StoreKey`s and `ModuleKey`s into a single interface so that modules have a single OCAPs handle
- code generation which makes inter-module communication more performant
- decoupling `ModuleKey` creation from `AppModuleBasic.Name()` so that app's can override root module account names
- inter-module hooks and plugins

## Alternatives

### **MsgServices vs `x/capability`**

The `x/capability` module does provide a proper object-capability implementation that can be used by any module in the Cosmos SDK and could even be used for inter-module OCAPs as described in [#5931](#).

The advantages of the approach described in this ADR are mostly around how it integrates with other parts of the Cosmos SDK, specifically:

- protobuf so that:
  - code generation of interfaces can be leveraged for a better dev UX
  - module interfaces are versioned and checked for breakage using [buf](#)
- sub-module accounts as per ADR 028
- the general `Msg` passing paradigm and the way signers are specified by `GetSigners`

Also, this is a complete replacement for keepers and could be applied to *all* inter-module communication whereas the `x/capability` approach in #5931 would need to be applied method by method.

## Consequences

### **Backwards Compatibility**

This ADR is intended to provide a pathway to a scenario where there is greater long term compatibility between modules. In the short-term, this will likely result in breaking certain `Keeper` interfaces which are too permissive and/or replacing `Keeper` interfaces altogether.

### **Positive**

- an alternative to keepers which can more easily lead to stable inter-module interfaces
- proper inter-module OCAPs
- improved module developer DevX, as commented on by several participants on [Architecture Review Call, Dec 3](#)
- lays the groundwork for what can be a greatly simplified `app.go`
- router can be setup to enforce atomic transactions for module-to-module calls

### **Negative**

- modules which adopt this will need significant refactoring

### **Neutral**

## Test Cases [optional]

## References

- [ADR 021](#)
- [ADR 031](#)
- [ADR 028](#)
- [ADR 030 draft](#)
- [Object-Capability Model](#)

# ADR 034: Account Rekeying

## Changelog

- 30-09-2020: Initial Draft

## Status

PROPOSED

## Abstract

Account rekeying is a process that allows an account to replace its authentication pubkey with a new one.

## Context

Currently, in the Cosmos SDK, the address of an auth `BaseAccount` is based on the hash of the public key. Once an account is created, the public key for the account is set in stone, and cannot be changed. This can be a problem for users, as key rotation is a useful security practice, but is not possible currently. Furthermore, as multisigs are a type of pubkey, once a multisig for an account is set, it can not be updated. This is problematic, as multisigs are often used by organizations or companies, who may need to change their set of multisig signers for internal reasons.

Transferring all the assets of an account to a new account with the updated pubkey is not sufficient, because some "engagements" of an account are not easily transferable. For example, in staking, to transfer bonded Atoms, an account would have to unbond all delegations and wait the three week unbonding period. Even more significantly, for validator operators, ownership over a validator is not transferrable at all, meaning that the operator key for a validator can never be updated, leading to poor operational security for validators.

## Decision

We propose the addition of a new feature to `x/auth` that allows accounts to update the public key associated with their account, while keeping the address the same.

This is possible because the Cosmos SDK `BaseAccount` stores the public key for an account in state, instead of making the assumption that the public key is included in the transaction (whether explicitly or implicitly through the signature) as in other blockchains such as Bitcoin and Ethereum. Because the public key is stored on chain, it is okay for the public key to not hash to the address of an account, as the address is not pertinent to the signature checking process.

To build this system, we design a new Msg type as follows:

```
service Msg {
 rpc ChangePubKey(MsgChangePubKey) returns (MsgChangePubKeyResponse);
}

message MsgChangePubKey {
 string address = 1;
 google.protobuf.Any pub_key = 2;
}

message MsgChangePubKeyResponse {}
```

The `MsgChangePubKey` transaction needs to be signed by the existing pubkey in state.

Once, approved, the handler for this message type, which takes in the `AccountKeeper`, will update the in-state pubkey for the account and replace it with the pubkey from the `Msg`.

An account that has had its pubkey changed cannot be automatically pruned from state. This is because if pruned, the original pubkey of the account would be needed to recreate the same address, but the owner of the address may not have the original pubkey anymore. Currently, we do not automatically prune any accounts anyways, but we would like to keep this option open the road (this is the purpose of account numbers). To resolve this, we charge an additional gas fee for this operation to compensate for this externality (this bound gas amount is configured as parameter `PubKeyChangeCost`). The bonus gas is charged inside the handler, using the `ConsumeGas` function. Furthermore, in the future, we can allow accounts that have rekeyed manually prune themselves using a new `Msg` type such as `MsgDeleteAccount`. Manually pruning accounts can give a gas refund as an incentive for performing the action.

```
amount := ak.GetParams(ctx).PubKeyChangeCost
ctx.GasMeter().ConsumeGas(amount, "pubkey change fee")
```

Everytime a key for an address is changed, we will store a log of this change in the state of the chain, thus creating a stack of all previous keys for an address and the time intervals for which they were active. This allows dapps and clients to easily query past keys for an account which may be useful for features such as verifying timestamped off-chain signed messages.

## Consequences

### Positive

- Will allow users and validator operators to employ better operational security practices with key rotation.
- Will allow organizations or groups to easily change and add/remove multisig signers.

### Negative

Breaks the current assumed relationship between address and pubkeys as  $H(\text{pubkey}) = \text{address}$ . This has a couple of consequences.

- This makes wallets that support this feature more complicated. For example, if an address on chain was updated, the corresponding key in the CLI wallet also needs to be updated.

- Cannot automatically prune accounts with 0 balance that have had their pubkey changed.

## Neutral

- While the purpose of this is intended to allow the owner of an account to update to a new pubkey they own, this could technically also be used to transfer ownership of an account to a new owner. For example, this could be used to sell a staked position without unbonding or an account that has vesting tokens. However, the friction of this is very high as this would essentially have to be done as a very specific OTC trade. Furthermore, additional constraints could be added to prevent accounts with Vesting tokens to use this feature.
- Will require that PubKeys for an account are included in the genesis exports.

## References

- <https://www.algorand.com/resources/blog/announcing-rekeying>

# ADR 035: Rosetta API Support

## Authors

- Jonathan Gimeno (@jgimeno)
- David Grierson (@senormonito)
- Alessio Treglia (@alessio)
- Frojdy Dymylja (@fdymylja)

## Changelog

- 2021-05-12: the external library [cosmos-rosetta-gateway](#) has been moved within the Cosmos SDK.

## Context

[Rosetta API](#) is an open-source specification and set of tools developed by Coinbase to standardise blockchain interactions.

Through the use of a standard API for integrating blockchain applications it will

- Be easier for a user to interact with a given blockchain
- Allow exchanges to integrate new blockchains quickly and easily
- Enable application developers to build cross-blockchain applications such as block explorers, wallets and dApps at considerably lower cost and effort.

## Decision

It is clear that adding Rosetta API support to the Cosmos SDK will bring value to all the developers and Cosmos SDK based chains in the ecosystem. How it is implemented is key.

The driving principles of the proposed design are:

1. **Extensibility:** it must be as riskless and painless as possible for application developers to set-up network configurations to expose Rosetta API-compliant services.
2. **Long term support:** This proposal aims to provide support for all the supported Cosmos SDK release series.
3. **Cost-efficiency:** Backporting changes to Rosetta API specifications from `master` to the various stable branches of Cosmos SDK is a cost that needs to be reduced.

We will achieve these delivering on these principles by the following:

1. There will be a package `rosetta/lib` for the implementation of the core Rosetta API features, particularly:
  - a. The types and interfaces (`Client`, `OfflineClient` ...), this separates design from implementation detail.
  - b. The `Server` functionality as this is independent of the Cosmos SDK version.
  - c. The `Online/OfflineNetwork`, which is not exported, and implements the rosetta API using the `Client` interface to query the node, build tx and so on.
  - d. The `errors` package to extend rosetta errors.
2. Due to differences between the Cosmos release series, each series will have its own specific implementation of `Client` interface.
3. There will be two options for starting an API service in applications:
  - a. API shares the application process
  - b. API-specific process.

## Architecture

### The External Repo

This section will describe the proposed external library, including the service implementation, plus the defined types and interfaces.

#### Server

`Server` is a simple `struct` that is started and listens to the port specified in the settings. This is meant to be used across all the Cosmos SDK versions that are actively supported.

The constructor follows:

```
func NewServer(settings Settings) (Server, error)
```

`Settings`, which are used to construct a new server, are the following:

```
// Settings define the rosetta server settings
type Settings struct {
 // Network contains the information regarding the network
 Network *types.NetworkIdentifier
 // Client is the online API handler
 Client crgtypes.Client
 // Listen is the address the handler will listen at
 Listen string
 // Offline defines if the rosetta service should be exposed in offline mode
 Offline bool
 // Retries is the number of readiness checks that will be attempted when
 // instantiating the handler
 // valid only for online API
 Retries int
 // RetryWait is the time that will be waited between retries
 RetryWait time.Duration
}
```

#### Types

Package types uses a mixture of rosetta types and custom defined type wrappers, that the client must parse and return while executing operations.

## Interfaces

Every SDK version uses a different format to connect (rpc, gRPC, etc), query and build transactions, we have abstracted this in what is the `Client` interface. The client uses rosetta types, whilst the `Online/OfflineNetwork` takes care of returning correctly parsed rosetta responses and errors.

Each Cosmos SDK release series will have their own `Client` implementations. Developers can implement their own custom `Client`'s as required.

```
// Client defines the API the client implementation should provide.
type Client interface {
 // Needed if the client needs to perform some action before connecting.
 Bootstrap() error
 // Ready checks if the servicer constraints for queries are satisfied
 // for example the node might still not be ready, it's useful in process
 // when the rosetta instance might come up before the node itself
 // the servicer must return nil if the node is ready
 Ready() error

 // Data API

 // Balances fetches the balance of the given address
 // if height is not nil, then the balance will be displayed
 // at the provided height, otherwise last block balance will be returned
 Balances(ctx context.Context, addr string, height *int64) ([]*types.Amount,
error)
 // BlockByHashAlt gets a block and its transaction at the provided height
 BlockByHash(ctx context.Context, hash string) (BlockResponse, error)
 // BlockByHeightAlt gets a block given its height, if height is nil then last
block is returned
 BlockByHeight(ctx context.Context, height *int64) (BlockResponse, error)
 // BlockTransactionsByHash gets the block, parent block and transactions
 // given the block hash.
 BlockTransactionsByHash(ctx context.Context, hash string)
(BlockTransactionsResponse, error)
 // BlockTransactionsByHash gets the block, parent block and transactions
 // given the block hash.
 BlockTransactionsByHeight(ctx context.Context, height *int64)
(BlockTransactionsResponse, error)
 // GetTx gets a transaction given its hash
 GetTx(ctx context.Context, hash string) (*types.Transaction, error)
 // GetUnconfirmedTx gets an unconfirmed Tx given its hash
 // NOTE(fdymylja): NOT IMPLEMENTED YET!
 GetUnconfirmedTx(ctx context.Context, hash string) (*types.Transaction, error)
 // Mempool returns the list of the current non confirmed transactions
 Mempool(ctx context.Context) ([]*types.TransactionIdentifier, error)
 // Peers gets the peers currently connected to the node
 Peers(ctx context.Context) ([]*types.Peer, error)
 // Status returns the node status, such as sync data, version etc
 Status(ctx context.Context) (*types.SyncStatus, error)

 // Construction API
```

```

// PostTx posts txBytes to the node and returns the transaction identifier plus
metadata related
// to the transaction itself.
PostTx(txBytes []byte) (res *types.TransactionIdentifier, meta
map[string]interface{}, err error)
// ConstructionMetadataFromOptions
ConstructionMetadataFromOptions(ctx context.Context, options
map[string]interface{}) (meta map[string]interface{}, err error)
OfflineClient
}

// OfflineClient defines the functionalities supported without having access to the
node
type OfflineClient interface {
 NetworkInformationProvider
 // SignedTx returns the signed transaction given the tx bytes (msgs) plus the
signatures
 SignedTx(ctx context.Context, txBytes []byte, sigs []*types.Signature)
(signedTxBytes []byte, err error)
 // TxOperationsAndSignersAccountIdentifiers returns the operations related to a
transaction and the account
 // identifiers if the transaction is signed
 TxOperationsAndSignersAccountIdentifiers(signed bool, hexBytes []byte) (ops
[]*types.Operation, signers []*types.AccountIdentifier, err error)
 // ConstructionPayload returns the construction payload given the request
 ConstructionPayload(ctx context.Context, req *types.ConstructionPayloadsRequest)
(resp *types.ConstructionPayloadsResponse, err error)
 // PreprocessOperationsToOptions returns the options given the preprocess
operations
 PreprocessOperationsToOptions(ctx context.Context, req
*types.ConstructionPreprocessRequest) (options map[string]interface{}, err error)
 // AccountIdentifierFromPublicKey returns the account identifier given the
public key
 AccountIdentifierFromPublicKey(pubKey *types.PublicKey)
(*types.AccountIdentifier, error)
}

```

## 2. Cosmos SDK Implementation

The Cosmos SDK implementation, based on version, takes care of satisfying the `Client` interface. In Stargate, Launchpad and 0.37, we have introduced the concept of `rosetta.Msg`, this message is not in the shared repository as the `sdk.Msg` type differs between Cosmos SDK versions.

The `rosetta.Msg` interface follows:

```

// Msg represents a cosmos-sdk message that can be converted from and to a rosetta
operation.
type Msg interface {
 sdk.Msg
 ToOperations(withStatus, hasError bool) []*types.Operation
}

```

```
 FromOperations(ops []*types.Operation) (sdk.Msg, error)
}
```

Hence developers who want to extend the rosetta set of supported operations just need to extend their module's `sdk.Msgs` with the `ToOperations` and `FromOperations` methods.

### 3. API service invocation

As stated at the start, application developers will have two methods for invocation of the Rosetta API service:

1. Shared process for both application and API
2. Standalone API service

#### Shared Process (Only Stargate)

Rosetta API service could run within the same execution process as the application. This would be enabled via `app.toml` settings, and if gRPC is not enabled the rosetta instance would be spinned in offline mode (tx building capabilities only).

#### Separate API service

Client application developers can write a new command to launch a Rosetta API server as a separate process too, using the `rosetta` command contained in the `/server/rosetta` package. Construction of the command depends on Cosmos SDK version. Examples can be found inside `simd` for stargate, and `contrib/rosetta/simapp` for other release series.

## Status

Proposed

## Consequences

### Positive

- Out-of-the-box Rosetta API support within Cosmos SDK.
- Blockchain interface standardisation

## References

- <https://www.rosetta-api.org/>

# ADR 036: Arbitrary Message Signature Specification

## Changelog

- 28/10/2020 - Initial draft

## Authors

- Antoine Herzog (@antoineherzog)
- Zaki Manian (@zmanian)
- Aleksandr Bezobchuk (alexanderbez) [1]

- Frojdi Dymylja (@fdymylja)

## Status

Draft

## Abstract

Currently, in the Cosmos SDK, there is no convention to sign arbitrary message like on Ethereum. We propose with this specification, for Cosmos SDK ecosystem, a way to sign and validate off-chain arbitrary messages.

This specification serves the purpose of covering every use case, this means that cosmos-sdk applications developers decide how to serialize and represent `Data` to users.

## Context

Having the ability to sign messages off-chain has proven to be a fundamental aspect of nearly any blockchain. The notion of signing messages off-chain has many added benefits such as saving on computational costs and reducing transaction throughput and overhead. Within the context of the Cosmos, some of the major applications of signing such data includes, but is not limited to, providing a cryptographic secure and verifiable means of proving validator identity and possibly associating it with some other framework or organization. In addition, having the ability to sign Cosmos messages with a Ledger or similar HSM device.

Further context and use cases can be found in the references links.

## Decision

The aim is being able to sign arbitrary messages, even using Ledger or similar HSM devices.

As a result signed messages should look roughly like Cosmos SDK messages but **must not** be a valid on-chain transaction. `chain-id`, `account_number` and `sequence` can all be assigned invalid values.

Cosmos SDK 0.40 also introduces a concept of "auth\_info" this can specify SIGN\_MODES.

A spec should include an `auth_info` that supports SIGN\_MODE\_DIRECT and SIGN\_MODE\_LEGACY\_AMINO.

Create the `offchain` proto definitions, we extend the auth module with `offchain` package to offer functionalities to verify and sign offline messages.

An offchain transaction follows these rules:

- the memo must be empty
- nonce, sequence number must be equal to 0
- chain-id must be equal to ""
- fee gas must be equal to 0
- fee amount must be an empty array

Verification of an offchain transaction follows the same rules as an onchain one, except for the spec differences highlighted above.

The first message added to the `offchain` package is `MsgSignData`.

`MsgSignData` allows developers to sign arbitrary bytes valid offchain only. Where `Signer` is the account address of the signer. `Data` is arbitrary bytes which can represent `text`, `files`, `object s`. It's applications developers decision how `Data` should be deserialized, serialized and the object it can represent in their context.

It's applications developers decision how `Data` should be treated, by treated we mean the serialization and deserialization process and the Object `Data` should represent.

Proto definition:

```
// MsgSignData defines an arbitrary, general-purpose, off-chain message
message MsgSignData {
 // Signer is the sdk.AccAddress of the message signer
 bytes Signer = 1 [(gogoproto.jsontag) = "signer", (gogoproto.casttype) =
"github.com/cosmos/cosmos-sdk/types.AccAddress"];
 // Data represents the raw bytes of the content that is signed (text, json, etc)
 bytes Data = 2 [(gogoproto.jsontag) = "data"];
}
```

Signed `MsgSignData` json example:

```
{
 "type": "cosmos-sdk/StdTx",
 "value": {
 "msg": [
 {
 "type": "sign(MsgSignData",
 "value": {
 "signer": "cosmos1hftz5ugqmpg9243xeegsqqav62f8hnywsjr4xr",
 "data": "cmFuZG9t"
 }
 }
],
 "fee": {
 "amount": [],
 "gas": "0"
 },
 "signatures": [
 {
 "pub_key": {
 "type": "tendermint/PubKeySecp256k1",
 "value": "AqnDSiRoFmTPfq97xxEb2VkJ/Hm28cPsqsZm9jEVsYK9"
 },
 "signature": "8y8i34qJakkjse9pOD2De+dnlc4KvFgh0wQpes4eydN66D9kv7cmCEouRrkka9t1W9cAkIL52ErB+6ye7X5aEc"
 }
],
 "memo": ""
 }
}
```

## Consequences

There is a specification on how messages, that are not meant to be broadcast to a live chain, should be formed.

### Backwards Compatibility

Backwards compatibility is maintained as this is a new message spec definition.

### Positive

- A common format that can be used by multiple applications to sign and verify off-chain messages.
- The specification is primitive which means it can cover every use case without limiting what is possible to fit inside it.
- It gives room for other off-chain messages specifications that aim to target more specific and common use cases such as off-chain-based authN/authZ layers [2].

### Negative

- Current proposal requires a fixed relationship between an account address and a public key.
- Doesn't work with multisig accounts.

## Further discussion

- Regarding security in `MsgSignData`, the developer using `MsgSignData` is in charge of making the content laying in `Data` non-replayable when, and if, needed.
- the offchain package will be further extended with extra messages that target specific use cases such as, but not limited to, authentication in applications, payment channels, L2 solutions in general.

## References

1. <https://github.com/cosmos/ics/pull/33>
2. [https://github.com/cosmos/cosmos-sdk/pull/7727#discussion\\_r515668204](https://github.com/cosmos/cosmos-sdk/pull/7727#discussion_r515668204)
3. <https://github.com/cosmos/cosmos-sdk/pull/7727#issuecomment-722478477>
4. <https://github.com/cosmos/cosmos-sdk/pull/7727#issuecomment-721062923>

# ADR 037: Governance split votes

## Changelog

- 2020/10/28: Intial draft

## Status

Accepted

## Abstract

This ADR defines a modification to the governance module that would allow a staker to split their votes into several voting options. For example, it could use 70% of its voting power to vote Yes and 30% of its voting power to vote No.

## Context

Currently, an address can cast a vote with only one options (Yes/No/Abstain/NoWithVeto) and use their full voting power behind that choice.

However, often times the entity owning that address might not be a single individual. For example, a company might have different stakeholders who want to vote differently, and so it makes sense to allow them to split their voting power. Another example use case is exchanges. Many centralized exchanges often stake a portion of their users' tokens in their custody. Currently, it is not possible for them to do "passthrough voting" and giving their users voting rights over their tokens. However, with this system, exchanges can poll their users for voting preferences, and then vote on-chain proportionally to the results of the poll.

## Decision

We modify the vote structs to be

```
type WeightedVoteOption struct {
 Option string
 Weight sdk.Dec
}

type Vote struct {
 ProposalID int64
 Voter sdk.Address
 Options []WeightedVoteOption
}
```

And for backwards compatibility, we introduce `MsgVoteWeighted` while keeping `MsgVote`.

```
type MsgVote struct {
 ProposalID int64
 Voter sdk.Address
 Option Option
}

type MsgVoteWeighted struct {
 ProposalID int64
 Voter sdk.Address
 Options []WeightedVoteOption
}
```

The `ValidateBasic` of a `MsgVoteWeighted` struct would require that

1. The sum of all the Rates is equal to 1.0
2. No Option is repeated

The governance tally function will iterate over all the options in a vote and add to the tally the result of the voter's voting power \* the rate for that option.

```

tally() {
 results := map[types.VoteOption]sdk.Dec

 for _, vote := range votes {
 for i, weightedOption := range vote.Options {
 results[weightedOption.Option] += getVotingPower(vote.voter) *
weightedOption.Weight
 }
 }
}

```

The CLI command for creating a multi-option vote would be as such:

```
simd tx gov vote 1 "yes=0.6,no=0.3,abstain=0.05,no_with_veto=0.05" --from mykey
```

To create a single-option vote a user can do either

```
simd tx gov vote 1 "yes=1" --from mykey
```

or

```
simd tx gov vote 1 yes --from mykey
```

to maintain backwards compatibility.

## Consequences

### Backwards Compatibility

- Previous VoteMsg types will remain the same and so clients will not have to update their procedure unless they want to support the WeightedVoteMsg feature.
- When querying a Vote struct from state, its structure will be different, and so clients wanting to display all voters and their respective votes will have to handle the new format and the fact that a single voter can have split votes.
- The result of querying the tally function should have the same API for clients.

### Positive

- Can make the voting process more accurate for addresses representing multiple stakeholders, often some of the largest addresses.

### Negative

- Is more complex than simple voting, and so may be harder to explain to users. However, this is mostly mitigated because the feature is opt-in.

### Neutral

- Relatively minor change to governance tally function.

## ADR 038: KVStore state listening

## Changelog

- 11/23/2020: Initial draft
- 10/06/2022: Introduce plugin system based on hashicorp/go-plugin
- 10/14/2022:
  - Add `ListenCommit`, flatten the state writes in a block to a single batch.
  - Remove listeners from cache stores, should only listen to `rootmulti.Store`.
  - Remove `HaltAppOnDeliveryError()`, the errors are propagated by default, the implementations should return nil if don't want to propagate errors.
- 26/05/2023: Update with ABCI 2.0

## Status

Proposed

## Abstract

This ADR defines a set of changes to enable listening to state changes of individual KVStores and exposing these data to consumers.

## Context

Currently, KVStore data can be remotely accessed through [Queries](#) which proceed either through Tendermint and the ABCI, or through the gRPC server. In addition to these request/response queries, it would be beneficial to have a means of listening to state changes as they occur in real time.

## Decision

We will modify the `CommitMultiStore` interface and its concrete (`rootmulti`) implementations and introduce a new `listenkv.Store` to allow listening to state changes in underlying KVStores. We don't need to listen to cache stores, because we can't be sure that the writes will be committed eventually, and the writes are duplicated in `rootmulti.Store` eventually, so we should only listen to `rootmulti.Store`. We will introduce a plugin system for configuring and running streaming services that write these state changes and their surrounding ABCI message context to different destinations.

## Listening

In a new file, `store/types/listening.go`, we will create a `MemoryListener` struct for streaming out protobuf encoded KV pairs state changes from a KVStore. The `MemoryListener` will be used internally by the concrete `rootmulti` implementation to collect state changes from KVStores.

```
// MemoryListener listens to the state writes and accumulate the records in memory.
type MemoryListener struct {
 stateCache []StoreKeyValuePair
}

// NewMemoryListener creates a listener that accumulate the state writes in memory.
func NewMemoryListener() *MemoryListener {
 return &MemoryListener{}
}
```

```

// OnWrite writes state change events to the internal cache
func (fl *MemoryListener) OnWrite(storeKey StoreKey, key []byte, value []byte,
delete bool) {
 fl.stateCache = append(fl.stateCache, StoreKeyValuePair{
 StoreKey: storeKey.Name(),
 Delete: delete,
 Key: key,
 Value: value,
 })
}

// PopStateCache returns the current state caches and set to nil
func (fl *MemoryListener) PopStateCache() []StoreKeyValuePair {
 res := fl.stateCache
 fl.stateCache = nil
 return res
}

```

We will also define a protobuf type for the KV pairs. In addition to the key and value fields this message will include the StoreKey for the originating KVStore so that we can collect information from separate KVStores and determine the source of each KV pair.

```

message StoreKeyValuePair {
 optional string store_key = 1; // the store key for the KVStore this pair
originates from
 required bool set = 2; // true indicates a set operation, false indicates a delete
operation
 required bytes key = 3;
 required bytes value = 4;
}

```

## ListenKVStore

We will create a new `Store` type `listenkv.Store` that the `rootmulti` store will use to wrap a `KVStore` to enable state listening. We will configure the `Store` with a `MemoryListener` which will collect state changes for output to specific destinations.

```

// Store implements the KVStore interface with listening enabled.
// Operations are traced on each core KVStore call and written to any of the
// underlying listeners with the proper key and operation permissions
type Store struct {
 parent types.KVStore
 listener *types.MemoryListener
 parentStoreKey types.StoreKey
}

// NewStore returns a reference to a new traceKVStore given a parent
// KVStore implementation and a buffered writer.
func NewStore(parent types.KVStore, psk types.StoreKey, listener

```

```

*types.MemoryListener) *Store {
 return &Store{parent: parent, listener: listener, parentStoreKey: psk}
}

// Set implements the KVStore interface. It traces a write operation and
// delegates the Set call to the parent KVStore.
func (s *Store) Set(key []byte, value []byte) {
 types.AssertValidKey(key)
 s.parent.Set(key, value)
 s.listener.OnWrite(s.parentStoreKey, key, value, false)
}

// Delete implements the KVStore interface. It traces a write operation and
// delegates the Delete call to the parent KVStore.
func (s *Store) Delete(key []byte) {
 s.parent.Delete(key)
 s.listener.OnWrite(s.parentStoreKey, key, nil, true)
}

```

## Multistore interface updates

We will update the `CommitMultiStore` interface to allow us to wrap a `MemoryListener` to a specific `KVStore`. Note that the `MemoryListener` will be attached internally by the concrete `rootmulti` implementation.

```

type CommitMultiStore interface {
 ...

 // AddListeners adds a listener for the KVStore belonging to the provided
 StoreKey
 AddListeners(keys []StoreKey)

 // PopStateCache returns the accumulated state change messages from
 MemoryListener
 PopStateCache() []StoreKeyValuePair
}

```

## Multistore implementation updates

We will adjust the `rootmulti GetKVStore` method to wrap the returned `KVStore` with a `listenkv.Store` if listening is turned on for that `Store`.

```

func (rs *Store) GetKVStore(key types.StoreKey) types.KVStore {
 store := rs.stores[key].(types.KVStore)

 if rs.TracingEnabled() {
 store = tracekv.NewStore(store, rs.traceWriter, rs.traceContext)
 }
 if rs.ListeningEnabled(key) {
 store = listenkv.NewStore(store, key, rs.listeners[key])
 }
}

```

```

 }

 return store
}

```

We will implement `AddListeners` to manage KVStore listeners internally and implement `PopStateCache` for a means of retrieving the current state.

```

// AddListeners adds state change listener for a specific KVStore
func (rs *Store) AddListeners(keys []types.StoreKey) {
 listener := types.NewMemoryListener()
 for i := range keys {
 rs.listeners[keys[i]] = listener
 }
}

```

```

func (rs *Store) PopStateCache() []types.StoreKeyValuePair {
 var cache []types.StoreKeyValuePair
 for _, ls := range rs.listeners {
 cache = append(cache, ls.PopStateCache()...)
 }
 sort.SliceStable(cache, func(i, j int) bool {
 return cache[i].StoreKey < cache[j].StoreKey
 })
 return cache
}

```

We will also adjust the `rootmulti.CacheMultiStore` and `CacheMultiStoreWithVersion` methods to enable listening in the cache layer.

```

func (rs *Store) CacheMultiStore() types.CacheMultiStore {
 stores := make(map[types.StoreKey]types.CacheWrapper)
 for k, v := range rs.stores {
 store := v.(types.KVStore)
 // Wire the listenkv.Store to allow listeners to observe the writes from the
 // cache store,
 // set same listeners on cache store will observe duplicated writes.
 if rs.ListeningEnabled(k) {
 store = listenkv.NewStore(store, k, rs.listeners[k])
 }
 stores[k] = store
 }
 return cachemulti.NewStore(rs.db, stores, rs.keysByName, rs.traceWriter,
 rs.getTracingContext())
}

```

```

func (rs *Store) CacheMultiStoreWithVersion(version int64) (types.CacheMultiStore,
 error) {
 // ...
}

```

```

 // Wire the listenkv.Store to allow listeners to observe the writes from the
 cache store,
 // set same listeners on cache store will observe duplicated writes.
 if rs.ListeningEnabled(key) {
 cacheStore = listenkv.NewStore(cacheStore, key, rs.listeners[key])
 }

 cachedStores[key] = cacheStore
}

return cachemulti.NewStore(rs.db, cachedStores, rs.keysByName, rs.traceWriter,
rs.getTracingContext()), nil
}

```

## Exposing the data

### Streaming Service

We will introduce a new `ABCIListener` interface that plugs into the `BaseApp` and relays ABCI requests and responses so that the service can group the state changes with the ABCI requests.

```

// baseapp/streaming.go

// ABCIListener is the interface that we're exposing as a streaming service.
type ABCIListener interface {
 // ListenFinalizeBlock updates the streaming service with the latest
 FinalizeBlock messages
 ListenFinalizeBlock(ctx context.Context, req abci.RequestFinalizeBlock, res
abci.ResponseFinalizeBlock) error
 // ListenCommit updates the streaming service with the latest Commit messages and
 state changes
 ListenCommit(ctx context.Context, res abci.ResponseCommit, changeSet
[*StoreKeyValuePair] error
}

```

### BaseApp Registration

We will add a new method to the `BaseApp` to enable the registration of `StreamingService`s:

```

// SetStreamingService is used to set a streaming service into the BaseApp hooks and
load the listeners into the multistore
func (app *BaseApp) SetStreamingService(s ABCIListener) {
 // register the StreamingService within the BaseApp
 // BaseApp will pass BeginBlock, DeliverTx, and EndBlock requests and responses
 to the streaming services to update their ABCI context
 app.abciListeners = append(app.abciListeners, s)
}

```

We will add two new fields to the `BaseApp` struct:

```

type BaseApp struct {
 ...
 // abcIListenersAsync for determining if abcIListeners will run asynchronously.
 // When abcIListenersAsync=false and stopNodeOnABCIListenerErr=false listeners
 // will run synchronized but will not stop the node.
 // When abcIListenersAsync=true stopNodeOnABCIListenerErr will be ignored.
 abcIListenersAsync bool

 // stopNodeOnABCIListenerErr halts the node when ABCI streaming service
 // listening results in an error.
 // stopNodeOnABCIListenerErr=true must be paired with abcIListenersAsync=false.
 stopNodeOnABCIListenerErr bool
}

```

## ABCI Event Hooks

We will modify the `FinalizeBlock` and `Commit` methods to pass ABCI requests and responses to any streaming service hooks registered with the `BaseApp`.

```

func (app *BaseApp) FinalizeBlock(req abci.RequestFinalizeBlock)
abci.ResponseFinalizeBlock {

 var abcRes abci.ResponseFinalizeBlock
 defer func() {
 // call the streaming service hook with the FinalizeBlock messages
 for _, abcListener := range app.abcIListeners {
 ctx := app.finalizeState.ctx
 blockHeight := ctx.BlockHeight()
 if app.abcIListenersAsync {
 go func(req abci.RequestFinalizeBlock, res
abci.ResponseFinalizeBlock) {
 if err := app.abcIListener.FinalizeBlock(blockHeight, req, res);
err != nil {
 app.logger.Error("FinalizeBlock listening hook failed",
"height", blockHeight, "err", err)
 }
 }(req, abcRes)
 } else {
 if err := app.abcIListener.ListenFinalizeBlock(blockHeight, req,
res); err != nil {
 app.logger.Error("FinalizeBlock listening hook failed",
"height", blockHeight, "err", err)
 if app.stopNodeOnABCIListenerErr {
 os.Exit(1)
 }
 }
 }
 }
 }()
}

```

```

 ...
 return abciRes
}

func (app *BaseApp) Commit() abci.ResponseCommit {
 ...

 res := abci.ResponseCommit{
 Data: commitID.Hash,
 RetainHeight: retainHeight,
 }

 // call the streaming service hook with the Commit messages
 for _, abciListener := range app.abciListeners {
 ctx := app.deliverState.ctx
 blockHeight := ctx.BlockHeight()
 changeSet := app.cms.PopStateCache()
 if app.abciListenersAsync {
 go func(res abci.ResponseCommit, changeSet []store.StoreKVPair) {
 if err := app.abciListener.ListenCommit(ctx, res, changeSet); err != nil {
 app.logger.Error("ListenCommit listening hook failed", "height",
blockHeight, "err", err)
 }
 }(res, changeSet)
 } else {
 if err := app.abciListener.ListenCommit(ctx, res, changeSet); err != nil {
 app.logger.Error("ListenCommit listening hook failed", "height",
blockHeight, "err", err)
 if app.stopNodeOnABCIListenerErr {
 os.Exit(1)
 }
 }
 }
 }

 ...
 return res
}

```

## Go Plugin System

We propose a plugin architecture to load and run `Streaming` plugins and other types of implementations. We will introduce a plugin system over gRPC that is used to load and run Cosmos-SDK plugins. The plugin system uses [hashicorp/go-plugin](#). Each plugin must have a struct that implements the `plugin.Plugin`

interface and an `Impl` interface for processing messages over gRPC. Each plugin must also have a message protocol defined for the gRPC service:

```
// streaming/plugins/abci/{plugin_version}/interface.go

// Handshake is a common handshake that is shared by streaming and host.
// This prevents users from executing bad plugins or executing a plugin
// directory. It is a UX feature, not a security feature.
var Handshake = plugin.HandshakeConfig{
 ProtocolVersion: 1,
 MagicCookieKey: "ABCI_LISTENER_PLUGIN",
 MagicCookieValue: "ef78114d-7bdf-411c-868f-347c99a78345",
}

// ListenerPlugin is the base struc for all kinds of go-plugin implementations
// It will be included in interfaces of different Plugins
type ABCIListenerPlugin struct {
 // GRPCPlugin must still implement the Plugin interface
 plugin.Plugin
 // Concrete implementation, written in Go. This is only used for plugins
 // that are written in Go.
 Impl baseapp.ABCIListener
}

func (p *ListenerGRPCPlugin) GRPCServer(_ *plugin.GRPCBroker, s *grpc.Server) error
{
 RegisterABCIListenerServiceServer(s, &GRPCServer{Impl: p.Impl})
 return nil
}

func (p *ListenerGRPCPlugin) GRPCCClient(
 _ context.Context,
 _ *plugin.GRPCBroker,
 c *grpc.ClientConn,
) (interface{}, error) {
 return &GRPCCClient{client: NewABCIListenerServiceClient(c)}, nil
}
```

The `plugin.Plugin` interface has two methods `Client` and `Server`. For our gRPC service these are `GRPCCClient` and `GRPCServer`. The `Impl` field holds the concrete implementation of our `baseapp.ABCIListener` interface written in Go. Note: this is only used for plugin implementations written in Go.

The advantage of having such a plugin system is that within each plugin authors can define the message protocol in a way that fits their use case. For example, when state change listening is desired, the `ABCIListener` message protocol can be defined as below (*for illustrative purposes only*). When state change listening is not desired than `ListenCommit` can be omitted from the protocol.

```
syntax = "proto3";
```

```

...
message Empty {}

message ListenFinalizeBlockRequest {
 RequestFinalizeBlock req = 1;
 ResponseFinalizeBlock res = 2;
}

message ListenCommitRequest {
 int64 block_height = 1;
 ResponseCommit res = 2;
 repeated StoreKVPair changeSet = 3;
}

// plugin that listens to state changes
service ABCIListenerService {
 rpc ListenFinalizeBlock(ListenFinalizeBlockRequest) returns (Empty);
 rpc ListenCommit(ListenCommitRequest) returns (Empty);
}

```

```

...
// plugin that doesn't listen to state changes
service ABCIListenerService {
 rpc ListenFinalizeBlock(ListenFinalizeBlockRequest) returns (Empty);
 rpc ListenCommit(ListenCommitRequest) returns (Empty);
}

```

Implementing the service above:

```

// streaming/plugins/abci/{plugin_version}/grpc.go

var (
 _ baseapp.ABCIListener = (*GRPCClient)(nil)
)

// GRPCClient is an implementation of the ABCIListener and ABCIListenerPlugin
// interfaces that talks over RPC.
type GRPCClient struct {
 client ABCIListenerServiceClient
}

func (m *GRPCClient) ListenFinalizeBlock(goCtx context.Context, req abci.RequestFinalizeBlock, res abci.ResponseFinalizeBlock) error {
 ctx := sdk.UnwrapSDKContext(goCtx)
 _, err := m.client.ListenDeliverTx(ctx, &ListenDeliverTxRequest{BlockHeight: ctx.BlockHeight(), Req: req, Res: res})
 return err
}

func (m *GRPCClient) ListenCommit(goCtx context.Context, res abci.ResponseCommit,

```

```

changeSet []store.StoreKVPair) error {
 ctx := sdk.UnwrapSDKContext(goCtx)
 _, err := m.client.ListenCommit(ctx, &ListenCommitRequest{BlockHeight:
ctx.BlockHeight(), Res: res, ChangeSet: changeSet})
 return err
}

// GRPCServer is the gRPC server that GRPCCClient talks to.
type GRPCServer struct {
 // This is the real implementation
 Impl baseapp.ABCIListener
}

func (m *GRPCServer) ListenFinalizeBlock(ctx context.Context, req
*ListenFinalizeBlockRequest) (*Empty, error) {
 return &Empty{}, m.Impl.ListenFinalizeBlock(ctx, req.Req, req.Res)
}

func (m *GRPCServer) ListenCommit(ctx context.Context, req *ListenCommitRequest)
(*Empty, error) {
 return &Empty{}, m.Impl.ListenCommit(ctx, req.Res, req.ChangeSet)
}

```

And the pre-compiled Go plugin `Impl` (*this is only used for plugins that are written in Go*):

```

// streaming/plugins/abci/{plugin_version}/impl/plugin.go

// Plugins are pre-compiled and loaded by the plugin system

// ABCIListener is the implementation of the baseapp.ABCIListener interface
type ABCIListener struct{}

func (m *ABCIListenerPlugin) ListenFinalizeBlock(ctx context.Context, req
abci.RequestFinalizeBlock, res abci.ResponseFinalizeBlock) error {
 // send data to external system
}

func (m *ABCIListenerPlugin) ListenCommit(ctx context.Context, res
abci.ResponseCommit, changeSet []store.StoreKVPair) error {
 // send data to external system
}

func main() {
 plugin.Serve(&plugin.ServeConfig{
 HandshakeConfig: grpc_abci_v1.Handshake,
 Plugins: map[string]plugin.Plugin{
 "grpc_plugin_v1": &grpc_abci_v1.ABCIListenerGRPCPlugin{Impl:
&ABCIListenerPlugin{}},
 },
 },
 // A non-nil value here enables gRPC serving for this streaming...
}

```

```

 GRPCServer: plugin.DefaultGRPCServer,
 })
}

```

We will introduce a plugin loading system that will return `(interface{}, error)`. This provides the advantage of using versioned plugins where the plugin interface and gRPC protocol change over time. In addition, it allows for building independent plugin that can expose different parts of the system over gRPC.

```

func NewStreamingPlugin(name string, logLevel string) (interface{}, error) {
 logger := hclog.New(&hclog.LoggerOptions{
 Output: hclog.DefaultOutput,
 Level: toHclogLevel(logLevel),
 Name: fmt.Sprintf("plugin.%s", name),
 })

 // We're a host. Start by launching the streaming process.
 env := os.Getenv(GetPluginEnvKey(name))
 client := plugin.NewClient(&plugin.ClientConfig{
 HandshakeConfig: HandshakeMap[name],
 Plugins: PluginMap,
 Cmd: exec.Command("sh", "-c", env),
 Logger: logger,
 AllowedProtocols: []plugin.Protocol{
 plugin.ProtocolNetRPC, plugin.ProtocolGRPC,
 },
 })

 // Connect via RPC
 rpcClient, err := client.Client()
 if err != nil {
 return nil, err
 }

 // Request streaming plugin
 return rpcClient.Dispense(name)
}

```

We propose a `RegisterStreamingPlugin` function for the App to register `NewStreamingPlugin`s with the App's `BaseApp`. Streaming plugins can be of `Any` type; therefore, the function takes in an interface vs a concrete type. For example, we could have plugins of `ABCIListener`, `WasmListener` or `IBCListener`. Note that `RegisterStreamingPluing` function is helper function and not a requirement. Plugin registration can easily be moved from the App to the `BaseApp` directly.

```

// baseapp/streaming.go

// RegisterStreamingPlugin registers streaming plugins with the App.
// This method returns an error if a plugin is not supported.
func RegisterStreamingPlugin(
 bApp *BaseApp,
 appOpts servertypes.AppOptions,
 keys map[string]*types.KVStoreKey,
)

```

```

 streamingPlugin interface{},
) error {
 switch t := streamingPlugin.(type) {
 case ABCIListener:
 registerABCIListenerPlugin(bApp, appOpts, keys, t)
 default:
 return fmt.Errorf("unexpected plugin type %T", t)
 }
 return nil
}

```

```

func registerABCIListenerPlugin(
 bApp *BaseApp,
 appOpts servertypes.AppOptions,
 keys map[string]*store.KVStoreKey,
 abcilistener ABCIListener,
) {
 asyncKey := fmt.Sprintf("%s.%s.%s", StreamingTomlKey, StreamingABCITomlKey,
 StreamingABCIAsync)
 async := cast.ToBoolean(appOpts.Get(asyncKey))
 stopNodeOnErrKey := fmt.Sprintf("%s.%s.%s", StreamingTomlKey,
 StreamingABCITomlKey, StreamingABCIStopNodeOnErrTomlKey)
 stopNodeOnErr := cast.ToBoolean(appOpts.Get(stopNodeOnErrKey))
 keysKey := fmt.Sprintf("%s.%s.%s", StreamingTomlKey, StreamingABCITomlKey,
 StreamingABCIKeysTomlKey)
 exposeKeysStr := cast.ToStringSlice(appOpts.Get(keysKey))
 exposedKeys := exposeStoreKeysSorted(exposeKeysStr, keys)
 bApp.cms.AddListeners(exposedKeys)
 bApp.SetStreamingService(abcilistener)
 bApp.stopNodeOnABCIListenerErr = stopNodeOnErr
 bApp.abcilistenersAsync = async
}

```

```

func exposeAll(list []string) bool {
 for _, ele := range list {
 if ele == "*" {
 return true
 }
 }
 return false
}

func exposeStoreKeys(keysStr []string, keys map[string]*types.KVStoreKey)
[]types.StoreKey {
 var exposeStoreKeys []types.StoreKey
 if exposeAll(keysStr) {
 exposeStoreKeys = make([]types.StoreKey, 0, len(keys))
 for _, storeKey := range keys {
 exposeStoreKeys = append(exposeStoreKeys, storeKey)
 }
 }
}

```

```

} else {
 exposeStoreKeys = make([]types.StoreKey, 0, len(keysStr))
 for _, keyStr := range keysStr {
 if storeKey, ok := keys[keyStr]; ok {
 exposeStoreKeys = append(exposeStoreKeys, storeKey)
 }
 }
}
// sort storeKeys for deterministic output
sort.SliceStable(exposeStoreKeys, func(i, j int) bool {
 return exposeStoreKeys[i].Name() < exposeStoreKeys[j].Name()
})

return exposeStoreKeys
}

```

The `NewStreamingPlugin` and `RegisterStreamingPlugin` functions are used to register a plugin with the App's `BaseApp`.

e.g. in `NewSimApp` :

```

func NewSimApp(
 logger log.Logger,
 db dbm.DB,
 traceStore io.Writer,
 loadLatest bool,
 appOpts servertypes.AppOptions,
 baseAppOptions ...func(*baseapp.BaseApp),
) *SimApp {
 ...
 keys := sdk.NewKVStoreKeys(
 authotypes.StoreKey, banktypes.StoreKey, stakingtypes.StoreKey,
 minttypes.StoreKey, distratypes.StoreKey, slashingtypes.StoreKey,
 govtypes.StoreKey, paramstypes.StoreKey, ibchost.StoreKey,
 upgradetypes.StoreKey,
 evidencetypes.StoreKey, ibctransfertypes.StoreKey, capabilitytypes.StoreKey,
)
 ...
 // register streaming services
 streamingCfg := cast.ToStringMap(appOpts.Get(baseapp.StreamingTomlKey))
 for service := range streamingCfg {
 pluginKey := fmt.Sprintf("%s.%s.%s", baseapp.StreamingTomlKey, service,
 baseapp.StreamingPluginTomlKey)
 pluginName := strings.TrimSpace(cast.ToString(appOpts.Get(pluginKey)))
 if len(pluginName) > 0 {
 LogLevel := cast.ToString(appOpts.Get(flags.FlagLogLevel))
 plugin, err := streaming.NewStreamingPlugin(pluginName, LogLevel)

```

```

 if err != nil {
 tmos.Exit(err.Error())
 }
 if err := baseapp.RegisterStreamingPlugin(bApp, appOpts, keys, plugin);
err != nil {
 tmos.Exit(err.Error())
}
}

return app

```

## Configuration

The plugin system will be configured within an App's TOML configuration files.

```

gRPC streaming
[streaming]

ABCI streaming service
[streaming.abci]

The plugin version to use for ABCI listening
plugin = "abci_v1"

List of kv store keys to listen to for state changes.
Set to ["*"] to expose all keys.
keys = ["*"]

Enable abciListeners to run asynchronously.
When abciListenersAsync=false and stopNodeOnABCIListenerErr=false listeners will
run synchronized but will not stop the node.
When abciListenersAsync=true stopNodeOnABCIListenerErr will be ignored.
async = false

Whether to stop the node on message deliver error.
stop-node-on-err = true

```

There will be four parameters for configuring `ABCIListener` plugin: `streaming.abci.plugin`, `streaming.abci.keys`, `streaming.abci.async` and `streaming.abci.stop-node-on-err`. `streaming.abci.plugin` is the name of the plugin we want to use for streaming, `streaming.abci.keys` is a set of store keys for stores it listens to, `streaming.abci.async` is bool enabling asynchronous listening and `streaming.abci.stop-node-on-err` is a bool that stops the node when true and when operating on synchronized mode `streaming.abci.async=false`. Note that `streaming.abci.stop-node-on-err=true` will be ignored if `streaming.abci.async=true`.

The configuration above support additional streaming plugins by adding the plugin to the `[streaming]` configuration section and registering the plugin with `RegisterStreamingPlugin` helper function.

Note the that each plugin must include `streaming.{service}.plugin` property as it is a requirement for doing the lookup and registration of the plugin with the App. All other properties are unique to the individual

services.

### Encoding and decoding streams

ADR-038 introduces the interfaces and types for streaming state changes out from KVStores, associating this data with their related ABCI requests and responses, and registering a service for consuming this data and streaming it to some destination in a final format. Instead of prescribing a final data format in this ADR, it is left to a specific plugin implementation to define and document this format. We take this approach because flexibility in the final format is necessary to support a wide range of streaming service plugins. For example, the data format for a streaming service that writes the data out to a set of files will differ from the data format that is written to a Kafka topic.

## Consequences

These changes will provide a means of subscribing to KVStore state changes in real time.

### Backwards Compatibility

- This ADR changes the `CommitMultiStore` interface, implementations supporting the previous version of this interface will not support the new one

### Positive

- Ability to listen to KVStore state changes in real time and expose these events to external consumers

### Negative

- Changes `CommitMultiStore` interface and its implementations

### Neutral

- Introduces additional- but optional- complexity to configuring and running a cosmos application
- If an application developer opts to use these features to expose data, they need to be aware of the ramifications/risks of that data exposure as it pertains to the specifics of their application

# ADR 039: Epoched Staking

## Changelog

- 10-Feb-2021: Initial Draft

## Authors

- Dev Ojha (@valardragon)
- Sunny Aggarwal (@sunny97)

## Status

Proposed

## Abstract

This ADR updates the proof of stake module to buffer the staking weight updates for a number of blocks before updating the consensus' staking weights. The length of the buffer is dubbed an epoch. The prior

functionality of the staking module is then a special case of the abstracted module, with the epoch being set to 1 block.

## Context

The current proof of stake module takes the design decision to apply staking weight changes to the consensus engine immediately. This means that delegations and unbonds get applied immediately to the validator set. This decision was primarily done as it was implementationally simplest, and because we at the time believed that this would lead to better UX for clients.

An alternative design choice is to allow buffering staking updates (delegations, unbonds, validators joining) for a number of blocks. This 'epoch'd proof of stake consensus provides the guarantee that the consensus weights for validators will not change mid-epoch, except in the event of a slash condition.

Additionally, the UX hurdle may not be as significant as was previously thought. This is because it is possible to provide users immediate acknowledgement that their bond was recorded and will be executed.

Furthermore, it has become clearer over time that immediate execution of staking events comes with limitations, such as:

- Threshold based cryptography. One of the main limitations is that because the validator set can change so regularly, it makes the running of multiparty computation by a fixed validator set difficult. Many threshold-based cryptographic features for blockchains such as randomness beacons and threshold decryption require a computationally-expensive DKG process (will take much longer than 1 block to create). To productively use these, we need to guarantee that the result of the DKG will be used for a reasonably long time. It wouldn't be feasible to rerun the DKG every block. By epoching staking, it guarantees we'll only need to run a new DKG once every epoch.
- Light client efficiency. This would lessen the overhead for IBC when there is high churn in the validator set. In the Tendermint light client bisection algorithm, the number of headers you need to verify is related to bounding the difference in validator sets between a trusted header and the latest header. If the difference is too great, you verify more header in between the two. By limiting the frequency of validator set changes, we can reduce the worst case size of IBC lite client proofs, which occurs when a validator set has high churn.
- Fairness of deterministic leader election. Currently we have no ways of reasoning of fairness of deterministic leader election in the presence of staking changes without epochs (tendermint/spec#217). Breaking fairness of leader election is profitable for validators, as they earn additional rewards from being the proposer. Adding epochs at least makes it easier for our deterministic leader election to match something we can prove secure. (Albeit, we still haven't proven if our current algorithm is fair with > 2 validators in the presence of stake changes)
- Staking derivative design. Currently, reward distribution is done lazily using the F1 fee distribution. While saving computational complexity, lazy accounting requires a more stateful staking implementation. Right now, each delegation entry has to track the time of last withdrawal. Handling this can be a challenge for some staking derivatives designs that seek to provide fungibility for all tokens staked to a single validator. Force-withdrawing rewards to users can help solve this, however it is infeasible to force-withdraw rewards to users on a per block basis. With epochs, a chain could more easily alter the design to have rewards be forcefully withdrawn (iterating over delegator accounts only once per-epoch), and can thus remove delegation timing from state. This may be useful for certain staking derivative designs.

## Design considerations

### Slashing

There is a design consideration for whether to apply a slash immediately or at the end of an epoch. A slash event should apply to only members who are actually staked during the time of the infraction, namely during the epoch the slash event occurred.

Applying it immediately can be viewed as offering greater consensus layer security, at potential costs to the aforementioned usecases. The benefits of immediate slashing for consensus layer security can be all be obtained by executing the validator jailing immediately (thus removing it from the validator set), and delaying the actual slash change to the validator's weight until the epoch boundary. For the use cases mentioned above, workarounds can be integrated to avoid problems, as follows:

- For threshold based cryptography, this setting will have the threshold cryptography use the original epoch weights, while consensus has an update that lets it more rapidly benefit from additional security. If the threshold based cryptography blocks liveness of the chain, then we have effectively raised the liveness threshold of the remaining validators for the rest of the epoch. (Alternatively, jailed nodes could still contribute shares) This plan will fail in the extreme case that more than 1/3rd of the validators have been jailed within a single epoch. For such an extreme scenario, the chain already have its own custom incident response plan, and defining how to handle the threshold cryptography should be a part of that.
- For light client efficiency, there can be a bit included in the header indicating an intra-epoch slash (ala <https://github.com/tendermint/spec/issues/199>).
- For fairness of deterministic leader election, applying a slash or jailing within an epoch would break the guarantee we were seeking to provide. This then re-introduces a new (but significantly simpler) problem for trying to provide fairness guarantees. Namely, that validators can adversarially elect to remove themselves from the set of proposers. From a security perspective, this could potentially be handled by two different mechanisms (or prove to still be too difficult to achieve). One is making a security statement acknowledging the ability for an adversary to force an ahead-of-time fixed threshold of users to drop out of the proposer set within an epoch. The second method would be to parameterize such that the cost of a slash within the epoch far outweighs benefits due to being a proposer. However, this latter criterion is quite dubious, since being a proposer can have many advantageous side-effects in chains with complex state machines. (Namely, DeFi games such as Fomo3D)
- For staking derivative design, there is no issue introduced. This does not increase the state size of staking records, since whether a slash has occurred is fully queryable given the validator address.

### Token lockup

When someone makes a transaction to delegate, even though they are not immediately staked, their tokens should be moved into a pool managed by the staking module which will then be used at the end of an epoch. This prevents concerns where they stake, and then spend those tokens not realizing they were already allocated for staking, and thus having their staking tx fail.

### Pipelining the epochs

For threshold based cryptography in particular, we need a pipeline for epoch changes. This is because when we are in epoch N, we want the epoch N+1 weights to be fixed so that the validator set can do the DKG accordingly. So if we are currently in epoch N, the stake weights for epoch N+1 should already be fixed, and new stake changes should be getting applied to epoch N + 2.

This can be handled by making a parameter for the epoch pipeline length. This parameter should not be alterable except during hard forks, to mitigate implementation complexity of switching the pipeline length.

With pipeline length 1, if I redelegate during epoch N, then my redelegation is applied prior to the beginning of epoch N+1. With pipeline length 2, if I redelegate during epoch N, then my redelegation is applied prior to the beginning of epoch N+2.

## Rewards

Even though all staking updates are applied at epoch boundaries, rewards can still be distributed immediately when they are claimed. This is because they do not affect the current stake weights, as we do not implement auto-bonding of rewards. If such a feature were to be implemented, it would have to be setup so that rewards are auto-bonded at the epoch boundary.

## Parameterizing the epoch length

When choosing the epoch length, there is a trade-off queued state/computation buildup, and countering the previously discussed limitations of immediate execution if they apply to a given chain.

Until an ABCI mechanism for variable block times is introduced, it is ill-advised to be using high epoch lengths due to the computation buildup. This is because when a block's execution time is greater than the expected block time from Tendermint, rounds may increment.

## Decision

**Step-1:** Implement buffering of all staking and slashing messages.

First we create a pool for storing tokens that are being bonded, but should be applied at the epoch boundary called the `EpochDelegationPool`. Then, we have two separate queues, one for staking, one for slashing. We describe what happens on each message being delivered below:

### Staking messages

- **MsgCreateValidator:** Move user's self-bond to `EpochDelegationPool` immediately. Queue a message for the epoch boundary to handle the self-bond, taking the funds from the `EpochDelegationPool`. If Epoch execution fail, return back funds from `EpochDelegationPool` to user's account.
- **MsgEditValidator:** Validate message and if valid queue the message for execution at the end of the Epoch.
- **MsgDelegate:** Move user's funds to `EpochDelegationPool` immediately. Queue a message for the epoch boundary to handle the delegation, taking the funds from the `EpochDelegationPool`. If Epoch execution fail, return back funds from `EpochDelegationPool` to user's account.
- **MsgBeginRedelegate:** Validate message and if valid queue the message for execution at the end of the Epoch.
- **MsgUndelegate:** Validate message and if valid queue the message for execution at the end of the Epoch.

### Slashing messages

- **MsgUnjail:** Validate message and if valid queue the message for execution at the end of the Epoch.
- **Slash Event:** Whenever a slash event is created, it gets queued in the slashing module to apply at the end of the epoch. The queues should be setup such that this slash applies immediately.

## Evidence Messages

- **MsgSubmitEvidence:** This gets executed immediately, and the validator gets jailed immediately. However in slashing, the actual slash event gets queued.

Then we add methods to the end blockers, to ensure that at the epoch boundary the queues are cleared and delegation updates are applied.

#### **Step-2:** Implement querying of queued staking txs.

When querying the staking activity of a given address, the status should return not only the amount of tokens staked, but also if there are any queued stake events for that address. This will require more work to be done in the querying logic, to trace the queued upcoming staking events.

As an initial implementation, this can be implemented as a linear search over all queued staking events. However, for chains that need long epochs, they should eventually build additional support for nodes that support querying to be able to produce results in constant time. (This is doable by maintaining an auxiliary hashmap for indexing upcoming staking events by address)

#### **Step-3:** Adjust gas

Currently gas represents the cost of executing a transaction when it's done immediately. (Merging together costs of p2p overhead, state access overhead, and computational overhead) However, now a transaction can cause computation in a future block, namely at the epoch boundary.

To handle this, we should initially include parameters for estimating the amount of future computation (denominated in gas), and add that as a flat charge needed for the message. We leave it as out of scope for how to weight future computation versus current computation in gas pricing, and have it set such that they are weighted equally for now.

## Consequences

### Positive

- Abstracts the proof of stake module that allows retaining the existing functionality
- Enables new features such as validator-set based threshold cryptography

### Negative

- Increases complexity of integrating more complex gas pricing mechanisms, as they now have to consider future execution costs as well.
- When epoch > 1, validators can no longer leave the network immediately, and must wait until an epoch boundary.

## ADR 040: Storage and SMT State Commitments

### Changelog

- 2020-01-15: Draft

### Status

DRAFT Not Implemented

### Abstract

Sparse Merkle Tree ([SMT](#)) is a version of a Merkle Tree with various storage and performance optimizations. This ADR defines a separation of state commitments from data storage and the Cosmos SDK transition from IAVL to SMT.

## Context

Currently, Cosmos SDK uses IAVL for both state [commitments](#) and data storage.

IAVL has effectively become an orphaned project within the Cosmos ecosystem and it's proven to be an inefficient state commitment data structure. In the current design, IAVL is used for both data storage and as a Merkle Tree for state commitments. IAVL is meant to be a standalone Merkelized key/value database, however it's using a KV DB engine to store all tree nodes. So, each node is stored in a separate record in the KV DB. This causes many inefficiencies and problems:

- Each object query requires a tree traversal from the root. Subsequent queries for the same object are cached on the Cosmos SDK level.
- Each edge traversal requires a DB query.
- Creating snapshots is [expensive](#). It takes about 30 seconds to export less than 100 MB of state (as of March 2020).
- Updates in IAVL may trigger tree reorganization and possible  $O(\log(n))$  hashes re-computation, which can become a CPU bottleneck.
- The node structure is pretty expensive - it contains a standard tree node elements (key, value, left and right element) and additional metadata such as height, version (which is not required by the Cosmos SDK). The entire node is hashed, and that hash is used as the key in the underlying database, [ref](#).

Moreover, the IAVL project lacks support and a maintainer and we already see better and well-established alternatives. Instead of optimizing the IAVL, we are looking into other solutions for both storage and state commitments.

## Decision

We propose to separate the concerns of state commitment (**SC**), needed for consensus, and state storage (**SS**), needed for state machine. Finally we replace IAVL with [Celestia's SMT](#). Celestia SMT is based on Diem (called jellyfish) design [\*] - it uses a compute-optimised SMT by replacing subtrees with only default values with a single node (same approach is used by Ethereum2) and implements compact proofs.

The storage model presented here doesn't deal with data structure nor serialization. It's a Key-Value database, where both key and value are binaries. The storage user is responsible for data serialization.

### Decouple state commitment from storage

Separation of storage and commitment (by the SMT) will allow the optimization of different components according to their usage and access patterns.

`SC` (SMT) is used to commit to a data and compute Merkle proofs. `SS` is used to directly access data. To avoid collisions, both `SS` and `SC` will use a separate storage namespace (they could use the same database underneath). `SS` will store each record directly (mapping `(key, value)` as `key → value`).

SMT is a merkle tree structure: we don't store keys directly. For every `(key, value)` pair, `hash(key)` is used as leaf path (we hash a key to uniformly distribute leaves in the tree) and `hash(value)` as the leaf contents. The tree structure is specified in more depth [below](#).

For data access we propose 2 additional KV buckets (implemented as namespaces for the key-value pairs, sometimes called [column family](#)):

1. B1: `key → value` : the principal object storage, used by a state machine, behind the Cosmos SDK `KVStore` interface: provides direct access by key and allows prefix iteration (KV DB backend must support it).
2. B2: `hash(key) → key` : a reverse index to get a key from an SMT path. Internally the SMT will `store (key, value) as prefix || hash(key) || hash(value)`. So, we can get an object value by composing `hash(key) → B2 → B1`.
3. We could use more buckets to optimize the app usage if needed.

We propose to use a KV database for both `ss` and `sc`. The store interface will allow to use the same physical DB backend for both `ss` and `sc` as well two separate DBs. The latter option allows for the separation of `ss` and `sc` into different hardware units, providing support for more complex setup scenarios and improving overall performance: one can use different backends (eg RocksDB and Badger) as well as independently tuning the underlying DB configuration.

## Requirements

State Storage requirements:

- range queries
- quick (key, value) access
- creating a snapshot
- historical versioning
- pruning (garbage collection)

State Commitment requirements:

- fast updates
- tree path should be short
- query historical commitment proofs using ICS-23 standard
- pruning (garbage collection)

## SMT for State Commitment

A Sparse Merkle tree is based on the idea of a complete Merkle tree of an intractable size. The assumption here is that as the size of the tree is intractable, there would only be a few leaf nodes with valid data blocks relative to the tree size, rendering a sparse tree.

The full specification can be found at [Celestia](#). In summary:

- The SMT consists of a binary Merkle tree, constructed in the same fashion as described in [Certificate Transparency \(RFC-6962\)](#), but using as the hashing function SHA-2-256 as defined in [FIPS 180-4](#).
- Leaves and internal nodes are hashed differently: the one-byte `0x00` is prepended for leaf nodes while `0x01` is prepended for internal nodes.
- Default values are given to leaf nodes with empty leaves.
- While the above rule is sufficient to pre-compute the values of intermediate nodes that are roots of empty subtrees, a further simplification is to extend this default value to all nodes that are roots of empty subtrees. The 32-byte zero is used as the default value. This rule takes precedence over the above one.

- An internal node that is the root of a subtree that contains exactly one non-empty leaf is replaced by that leaf's leaf node.

## Snapshots for storage sync and state versioning

Below, with simple *snapshot* we refer to a database snapshot mechanism, not to a *ABCI snapshot sync*. The latter will be referred as *snapshot sync* (which will directly use DB snapshot as described below).

Database snapshot is a view of DB state at a certain time or transaction. It's not a full copy of a database (it would be too big). Usually a snapshot mechanism is based on a *copy on write* and it allows DB state to be efficiently delivered at a certain stage. Some DB engines support snapshotting. Hence, we propose to reuse that functionality for the state sync and versioning (described below). We limit the supported DB engines to ones which efficiently implement snapshots. In a final section we discuss the evaluated DBs.

One of the Stargate core features is a *snapshot sync* delivered in the `/snapshot` package. It provides a way to trustlessly sync a blockchain without repeating all transactions from the genesis. This feature is implemented in Cosmos SDK and requires storage support. Currently IAVL is the only supported backend. It works by streaming to a client a snapshot of a `ss` at a certain version together with a header chain.

A new database snapshot will be created in every `EndBlocker` and identified by a block height. The `root store` keeps track of the available snapshots to offer `ss` at a certain version. The `root store` implements the `RootStore` interface described below. In essence, `RootStore` encapsulates a `Committer` interface. `Committer` has a `Commit`, `SetPruning`, `GetPruning` functions which will be used for creating and removing snapshots. The `rootStore.Commit` function creates a new snapshot and increments the version on each call, and checks if it needs to remove old versions. We will need to update the SMT interface to implement the `Committer` interface. NOTE: `Commit` must be called exactly once per block. Otherwise we risk going out of sync for the version number and block height. NOTE: For the Cosmos SDK storage, we may consider splitting that interface into `Committer` and `PruningCommitter` - only the multiroot should implement `PruningCommitter` (cache and prefix store don't need pruning).

Number of historical versions for `abci.RequestQuery` and state sync snapshots is part of a node configuration, not a chain configuration (configuration implied by the blockchain consensus). A configuration should allow to specify number of past blocks and number of past blocks modulo some number (eg: 100 past blocks and one snapshot every 100 blocks for past 2000 blocks). Archival nodes can keep all past versions.

Pruning old snapshots is effectively done by a database. Whenever we update a record in `sc`, SMT won't update nodes - instead it creates new nodes on the update path, without removing the old one. Since we are snapshotting each block, we need to change that mechanism to immediately remove orphaned nodes from the database. This is a safe operation - snapshots will keep track of the records and make it available when accessing past versions.

To manage the active snapshots we will either use a DB *max number of snapshots* option (if available), or we will remove DB snapshots in the `EndBlocker`. The latter option can be done efficiently by identifying snapshots with block height and calling a store function to remove past versions.

### Accessing old state versions

One of the functional requirements is to access old state. This is done through `abci.RequestQuery` structure. The version is specified by a block height (so we query for an object by a key `K` at block height `H`). The number of old versions supported for `abci.RequestQuery` is configurable. Accessing an old state is done by using available snapshots. `abci.RequestQuery` doesn't need old state of `sc` unless the

`prove=true` parameter is set. The SMT merkle proof must be included in the `abci.ResponseQuery` only if both `sc` and `ss` have a snapshot for requested version.

Moreover, Cosmos SDK could provide a way to directly access a historical state. However, a state machine shouldn't do that - since the number of snapshots is configurable, it would lead to nondeterministic execution.

We positively [validated](#) a versioning and snapshot mechanism for querying old state with regards to the database we evaluated.

## State Proofs

For any object stored in State Store (SS), we have corresponding object in `sc`. A proof for object `v` identified by a key `k` is a branch of `sc`, where the path corresponds to the key `hash(k)`, and the leaf is `hash(k, v)`.

## Rollbacks

We need to be able to process transactions and roll-back state updates if a transaction fails. This can be done in the following way: during transaction processing, we keep all state change requests (writes) in a `CacheWrapper` abstraction (as it's done today). Once we finish the block processing, in the `Endblocker`, we commit a root store - at that time, all changes are written to the SMT and to the `ss` and a snapshot is created.

## Committing to an object without saving it

We identified use-cases, where modules will need to save an object commitment without storing an object itself. Sometimes clients are receiving complex objects, and they have no way to prove a correctness of that object without knowing the storage layout. For those use cases it would be easier to commit to the object without storing it directly.

## Refactor MultiStore

The Stargate `/store` implementation (store/v1) adds an additional layer in the SDK store construction - the `MultiStore` structure. The multistore exists to support the modularity of the Cosmos SDK - each module is using its own instance of IAVL, but in the current implementation, all instances share the same database. The latter indicates, however, that the implementation doesn't provide true modularity. Instead it causes problems related to race condition and atomic DB commits (see: [#6370](#) and [discussion](#)).

We propose to reduce the multistore concept from the SDK, and to use a single instance of `sc` and `ss` in a `RootStore` object. To avoid confusion, we should rename the `MultiStore` interface to `RootStore`. The `RootStore` will have the following interface; the methods for configuring tracing and listeners are omitted for brevity.

```
// Used where read-only access to versions is needed.
type BasicRootStore interface {
 Store
 GetKVStore(StoreKey) KVStore
 CacheRootStore() CacheRootStore
}

// Used as the main app state, replacing CommitMultiStore.
```

```

type CommitRootStore interface {
 BasicRootStore
 Committer
 Snapshotter

 GetVersion(uint64) (BasicRootStore, error)
 SetInitialVersion(uint64) error

 ... // Trace and Listen methods
}

// Replaces CacheMultiStore for branched state.
type CacheRootStore interface {
 BasicRootStore
 Write()

 ... // Trace and Listen methods
}

// Example of constructor parameters for the concrete type.
type RootStoreConfig struct {
 Upgrades *StoreUpgrades
 InitialVersion uint64

 ReservePrefix(StoreKey, StoreType)
}

```

In contrast to `MultiStore`, `RootStore` doesn't allow to dynamically mount sub-stores or provide an arbitrary backing DB for individual sub-stores.

NOTE: modules will be able to use a special commitment and their own DBs. For example: a module which will use ZK proofs for state can store and commit this proof in the `RootStore` (usually as a single record) and manage the specialized store privately or using the `SC` low level interface.

### **Compatibility support**

To ease the transition to this new interface for users, we can create a shim which wraps a `CommitMultiStore` but provides a `CommitRootStore` interface, and expose functions to safely create and access the underlying `CommitMultiStore`.

The new `RootStore` and supporting types can be implemented in a `store/v2alpha1` package to avoid breaking existing code.

### **Merkle Proofs and IBC**

Currently, an IBC (v1.0) Merkle proof path consists of two elements (`["<store-key>", "<record-key>"]`), with each key corresponding to a separate proof. These are each verified according to individual [ICS-23 specs](#), and the result hash of each step is used as the committed value of the next step, until a root commitment hash is obtained. The root hash of the proof for `"<record-key>"` is hashed with the `"<store-key>"` to validate against the App Hash.

This is not compatible with the `RootStore`, which stores all records in a single Merkle tree structure, and won't produce separate proofs for the store- and record-key. Ideally, the store-key component of the proof

could just be omitted, and updated to use a "no-op" spec, so only the record-key is used. However, because the IBC verification code hardcodes the `"ibc"` prefix and applies it to the SDK proof as a separate element of the proof path, this isn't possible without a breaking change. Breaking this behavior would severely impact the Cosmos ecosystem which already widely adopts the IBC module. Requesting an update of the IBC module across the chains is a time consuming effort and not easily feasible.

As a workaround, the `RootStore` will have to use two separate SMTs (they could use the same underlying DB): one for IBC state and one for everything else. A simple Merkle map that reference these SMTs will act as a Merkle Tree to create a final App hash. The Merkle map is not stored in a DBs - it's constructed in the runtime. The IBC substore key must be `"ibc"`.

The workaround can still guarantee atomic syncs: the [proposed DB backends](#) support atomic transactions and efficient rollbacks, which will be used in the commit phase.

The presented workaround can be used until the IBC module is fully upgraded to supports single-element commitment proofs.

### Optimization: compress module key prefixes

We consider a compression of prefix keys by creating a mapping from module key to an integer, and serializing the integer using varint coding. Varint coding assures that different values don't have common byte prefix. For Merkle Proofs we can't use prefix compression - so it should only apply for the `SS` keys. Moreover, the prefix compression should be only applied for the module namespace. More precisely:

- each module has it's own namespace;
- when accessing a module namespace we create a KVStore with embedded prefix;
- that prefix will be compressed only when accessing and managing `SS`.

We need to assure that the codes won't change. We can fix the mapping in a static variable (provided by an app) or SS state under a special key.

TODO: need to make decision about the key compression.

### Optimization: SS key compression

Some objects may be saved with key, which contains a Protobuf message type. Such keys are long. We could save a lot of space if we can map Protobuf message types in varints.

TODO: finalize this or move to another ADR.

## Migration

Using the new store will require a migration. 2 Migrations are proposed:

1. Genesis export -- it will reset the blockchain history.
2. In place migration: we can reuse `UpgradeKeeper.SetUpgradeHandler` to provide the migration logic:

```
app.UpgradeKeeper.SetUpgradeHandler("adr-40", func(ctx sdk.Context, plan
upgradetypes.Plan, vm module.VersionMap) (module.VersionMap, error) {

 storev2.Migrate(iavlstore, v2.store)
```

```

 // RunMigrations returns the VersionMap
 // with the updated module ConsensusVersions
 return app.mm.RunMigrations(ctx, vm)
}

```

The `Migrate` function will read all entries from a store/v1 DB and save them to the AD-40 combined KV store. Cache layer should not be used and the operation must finish with a single Commit call.

Inserting records to the `SC` (SMT) component is the bottleneck. Unfortunately SMT doesn't support batch transactions. Adding batch transactions to `SC` layer is considered as a feature after the main release.

## Consequences

### Backwards Compatibility

This ADR doesn't introduce any Cosmos SDK level API changes.

We change the storage layout of the state machine, a storage hard fork and network upgrade is required to incorporate these changes. SMT provides a merkle proof functionality, however it is not compatible with ICS23. Updating the proofs for ICS23 compatibility is required.

### Positive

- Decoupling state from state commitment introduce better engineering opportunities for further optimizations and better storage patterns.
- Performance improvements.
- Joining SMT based camp which has wider and proven adoption than IAVL. Example projects which decided on SMT: Ethereum2, Diem (Libra), Trillan, Tezos, Celestia.
- Multistore removal fixes a longstanding issue with the current MultiStore design.
- Simplifies merkle proofs - all modules, except IBC, have only one pass for merkle proof.

### Negative

- Storage migration
- LL SMT doesn't support pruning - we will need to add and test that functionality.
- `SS` keys will have an overhead of a key prefix. This doesn't impact `SC` because all keys in `SC` have same size (they are hashed).

### Neutral

- Deprecating IAVL, which is one of the core proposals of Cosmos Whitepaper.

## Alternative designs

Most of the alternative designs were evaluated in [state commitments and storage report](#).

Ethereum research published [Verkle Trie](#) - an idea of combining polynomial commitments with merkle tree in order to reduce the tree height. This concept has a very good potential, but we think it's too early to implement it. The current, SMT based design could be easily updated to the Verkle Trie once other research implement all necessary libraries. The main advantage of the design described in this ADR is the separation of state commitments from the data storage and designing a more powerful interface.

## Further Discussions

## Evaluated KV Databases

We verified existing databases KV databases for evaluating snapshot support. The following databases provide efficient snapshot mechanism: Badger, RocksDB, [Pebble](#). Databases which don't provide such support or are not production ready: boltdb, leveldb, goleveldb, membdb, lmdb.

## RDBMS

Use of RDBMS instead of simple KV store for state. Use of RDBMS will require a Cosmos SDK API breaking change (`KVStore` interface) and will allow better data extraction and indexing solutions. Instead of saving an object as a single blob of bytes, we could save it as record in a table in the state storage layer, and as a `hash(key, protobuf(object))` in the SMT as outlined above. To verify that an object registered in RDBMS is same as the one committed to SMT, one will need to load it from RDBMS, marshal using protobuf, hash and do SMT search.

## Off Chain Store

We were discussing use case where modules can use a support database, which is not automatically committed. Module will responsible for having a sound storage model and can optionally use the feature discussed in *\_Committing to an object without saving it* section.

## References

- [IAVL What's Next?](#)
- [IAVL overview](#) of it's state v0.15
- [State commitments and storage report](#)
- [Celestia \(LazyLedger\) SMT](#)
- Facebook Diem (Libra) SMT [design](#)
- [Trillian Revocation Transparency, Trillian Verifiable Data Structures](#).
- Design and implementation [discussion](#).
- [How to Upgrade IBC Chains and their Clients](#)
- [ADR-40 Effect on IBC](#)

# ADR 041: In-Place Store Migrations

## Changelog

- 17.02.2021: Initial Draft

## Status

Accepted

## Abstract

This ADR introduces a mechanism to perform in-place state store migrations during chain software upgrades.

## Context

When a chain upgrade introduces state-breaking changes inside modules, the current procedure consists of exporting the whole state into a JSON file (via the `simd export` command), running migration scripts on the JSON file (`simd genesis migrate` command), clearing the stores (`simd unsafe-reset-all` command), and starting a new chain with the migrated JSON file as new genesis (optionally with a custom initial block height). An example of such a procedure can be seen [in the Cosmos Hub 3->4 migration guide](#).

This procedure is cumbersome for multiple reasons:

- The procedure takes time. It can take hours to run the `export` command, plus some additional hours to run `InitChain` on the fresh chain using the migrated JSON.
- The exported JSON file can be heavy (~100MB-1GB), making it difficult to view, edit and transfer, which in turn introduces additional work to solve these problems (such as [streaming genesis](#)).

## Decision

We propose a migration procedure based on modifying the KV store in-place without involving the JSON export-process-import flow described above.

### Module `ConsensusVersion`

We introduce a new method on the `AppModule` interface:

```
type AppModule interface {
 // --snip--
 ConsensusVersion() uint64
}
```

This methods returns an `uint64` which serves as state-breaking version of the module. It MUST be incremented on each consensus-breaking change introduced by the module. To avoid potential errors with default values, the initial version of a module MUST be set to 1. In the Cosmos SDK, version 1 corresponds to the modules in the v0.41 series.

### Module-Specific Migration Functions

For each consensus-breaking change introduced by the module, a migration script from `ConsensusVersion N` to version `N+1` MUST be registered in the `Configurator` using its newly-added `RegisterMigration` method. All modules receive a reference to the configurator in their `RegisterServices` method on `AppModule`, and this is where the migration functions should be registered. The migration functions should be registered in increasing order.

```
func (am AppModule) RegisterServices(cfg module.Configurator) {
 // --snip--
 cfg.RegisterMigration(types.ModuleName, 1, func(ctx sdk.Context) error {
 // Perform in-place store migrations from ConsensusVersion 1 to 2.
 })
 cfg.RegisterMigration(types.ModuleName, 2, func(ctx sdk.Context) error {
 // Perform in-place store migrations from ConsensusVersion 2 to 3.
 })
 // etc.
}
```

For example, if the new ConsensusVersion of a module is `N`, then `N-1` migration functions MUST be registered in the configurator.

In the Cosmos SDK, the migration functions are handled by each module's keeper, because the keeper holds the `sdk.StoreKey` used to perform in-place store migrations. To not overload the keeper, a `Migrator` wrapper is used by each module to handle the migration functions:

```
// Migrator is a struct for handling in-place store migrations.
type Migrator struct {
 BaseKeeper
}
```

Migration functions should live inside the `migrations/` folder of each module, and be called by the Migrator's methods. We propose the format `Migrate{M}to{N}` for method names.

```
// Migrate1to2 migrates from version 1 to 2.
func (m Migrator) Migrate1to2(ctx sdk.Context) error {
 return v2bank.MigrateStore(ctx, m.keeper.storeKey) // v043bank is package
 `x/bank/migrations/v2`.
}
```

Each module's migration functions are specific to the module's store evolutions, and are not described in this ADR. An example of x/bank store key migrations after the introduction of ADR-028 length-prefixed addresses can be seen in this [store.go code](#).

## Tracking Module Versions in `x/upgrade`

We introduce a new prefix store in `x/upgrade`'s store. This store will track each module's current version, it can be modeled as a `map[string]uint64` of module name to module ConsensusVersion, and will be used when running the migrations (see next section for details). The key prefix used is `0x1`, and the key/value format is:

```
0x2 | {bytes(module_name)} => BigEndian(module_consensus_version)
```

The initial state of the store is set from `app.go`'s `InitChainer` method.

The UpgradeHandler signature needs to be updated to take a `VersionMap`, as well as return an upgraded `VersionMap` and an error:

```
- type UpgradeHandler func(ctx sdk.Context, plan Plan)
+ type UpgradeHandler func(ctx sdk.Context, plan Plan, versionMap VersionMap)
(VersionMap, error)
```

To apply an upgrade, we query the `VersionMap` from the `x/upgrade` store and pass it into the handler. The handler runs the actual migration functions (see next section), and if successful, returns an updated `VersionMap` to be stored in state.

```
func (k UpgradeKeeper) ApplyUpgrade(ctx sdk.Context, plan types.Plan) {
 // --snip--
```

```

- handler(ctx, plan)
+ updatedVM, err := handler(ctx, plan, k.GetModuleVersionMap(ctx)) //
k.GetModuleVersionMap() fetches the VersionMap stored in state.
+ if err != nil {
+ return err
+ }
+
+ // Set the updated consensus versions to state
+ k.SetModuleVersionMap(ctx, updatedVM)
}

```

A gRPC query endpoint to query the `versionMap` stored in `x/upgrade`'s state will also be added, so that app developers can double-check the `VersionMap` before the upgrade handler runs.

## Running Migrations

Once all the migration handlers are registered inside the configurator (which happens at startup), running migrations can happen by calling the `RunMigrations` method on `module.Manager`. This function will loop through all modules, and for each module:

- Get the old ConsensusVersion of the module from its `VersionMap` argument (let's call it `M`).
- Fetch the new ConsensusVersion of the module from the `ConsensusVersion()` method on `AppModule` (call it `N`).
- If `N > M`, run all registered migrations for the module sequentially `M -> M+1 -> M+2...` until `N`.
  - There is a special case where there is no ConsensusVersion for the module, as this means that the module has been newly added during the upgrade. In this case, no migration function is run, and the module's current ConsensusVersion is saved to `x/upgrade`'s store.

If a required migration is missing (e.g. if it has not been registered in the `Configurator`), then the `RunMigrations` function will error.

In practice, the `RunMigrations` method should be called from inside an `UpgradeHandler`.

```

app.UpgradeKeeper.SetUpgradeHandler("my-plan", func(ctx sdk.Context, plan
upgradetypes.Plan, vm module.VersionMap) (module.VersionMap, error) {
 return app.mm.RunMigrations(ctx, vm)
})

```

Assuming a chain upgrades at block `n`, the procedure should run as follows:

- the old binary will halt in `BeginBlock` when starting block `N`. In its store, the ConsensusVersions of the old binary's modules are stored.
- the new binary will start at block `N`. The `UpgradeHandler` is set in the new binary, so will run at `BeginBlock` of the new binary. Inside `x/upgrade`'s `ApplyUpgrade`, the `VersionMap` will be retrieved from the (old binary's) store, and passed into the `RunMigrations` function, migrating all module stores in-place before the modules' own `BeginBlock`s.

## Consequences

## Backwards Compatibility

This ADR introduces a new method `ConsensusVersion()` on `AppModule`, which all modules need to implement. It also alters the `UpgradeHandler` function signature. As such, it is not backwards-compatible.

While modules MUST register their migration functions when bumping ConsensusVersions, running those scripts using an upgrade handler is optional. An application may perfectly well decide to not call the `RunMigrations` inside its upgrade handler, and continue using the legacy JSON migration path.

## Positive

- Perform chain upgrades without manipulating JSON files.
- While no benchmark has been made yet, it is probable that in-place store migrations will take less time than JSON migrations. The main reason supporting this claim is that both the `simd export` command on the old binary and the `InitChain` function on the new binary will be skipped.

## Negative

- Module developers MUST correctly track consensus-breaking changes in their modules. If a consensus-breaking change is introduced in a module without its corresponding `ConsensusVersion()` bump, then the `RunMigrations` function won't detect the migration, and the chain upgrade might be unsuccessful. Documentation should clearly reflect this.

## Neutral

- The Cosmos SDK will continue to support JSON migrations via the existing `simd export` and `simd genesis migrate` commands.
- The current ADR does not allow creating, renaming or deleting stores, only modifying existing store keys and values. The Cosmos SDK already has the `StoreLoader` for those operations.

## Further Discussions

## References

- Initial discussion: <https://github.com/cosmos/cosmos-sdk/discussions/8429>
- Implementation of `ConsensusVersion` and `RunMigrations` :  
<https://github.com/cosmos/cosmos-sdk/pull/8485>
- Issue discussing `x/upgrade` design: <https://github.com/cosmos/cosmos-sdk/issues/8514>

# ADR 042: Group Module

## Changelog

- 2020/04/09: Initial Draft

## Status

Draft

## Abstract

This ADR defines the `x/group` module which allows the creation and management of on-chain multi-signature accounts and enables voting for message execution based on configurable decision policies.

## Context

The legacy amino multi-signature mechanism of the Cosmos SDK has certain limitations:

- Key rotation is not possible, although this can be solved with [account rekeying](#).
- Thresholds can't be changed.
- UX is cumbersome for non-technical users ([#5661](#)).
- It requires `legacy_amino` sign mode ([#8141](#)).

While the group module is not meant to be a total replacement for the current multi-signature accounts, it provides a solution to the limitations described above, with a more flexible key management system where keys can be added, updated or removed, as well as configurable thresholds. It's meant to be used with other access control modules such as `x/feegrant` and `x/authz` to simplify key management for individuals and organizations.

The proof of concept of the group module can be found in <https://github.com/regen-network/regen-ledger/tree/master/proto/regen/group/v1alpha1> and <https://github.com/regen-network/regen-ledger/tree/master/x/group>.

## Decision

We propose merging the `x/group` module with its supporting [ORM/Table Store package \(#7098\)](#) into the Cosmos SDK and continuing development here. There will be a dedicated ADR for the ORM package.

### Group

A group is a composition of accounts with associated weights. It is not an account and doesn't have a balance. It doesn't in and of itself have any sort of voting or decision weight. Group members can create proposals and vote on them through group accounts using different decision policies.

It has an `admin` account which can manage members in the group, update the group metadata and set a new admin.

```
message GroupInfo {

 // group_id is the unique ID of this group.
 uint64 group_id = 1;

 // admin is the account address of the group's admin.
 string admin = 2;

 // metadata is any arbitrary metadata to attached to the group.
 bytes metadata = 3;

 // version is used to track changes to a group's membership structure that
 // would break existing proposals. Whenever a member weight has changed,
 // or any member is added or removed, the version is incremented and will
 // invalidate all proposals from older versions.
 uint64 version = 4;

 // total_weight is the sum of the group members' weights.

```

```

 string total_weight = 5;
}

message GroupMember {

 // group_id is the unique ID of the group.
 uint64 group_id = 1;

 // member is the member data.
 Member member = 2;
}

// Member represents a group member with an account address,
// non-zero weight and metadata.
message Member {

 // address is the member's account address.
 string address = 1;

 // weight is the member's voting weight that should be greater than 0.
 string weight = 2;

 // metadata is any arbitrary metadata to attached to the member.
 bytes metadata = 3;
}

```

## Group Account

A group account is an account associated with a group and a decision policy. A group account does have a balance.

Group accounts are abstracted from groups because a single group may have multiple decision policies for different types of actions. Managing group membership separately from decision policies results in the least overhead and keeps membership consistent across different policies. The pattern that is recommended is to have a single master group account for a given group, and then to create separate group accounts with different decision policies and delegate the desired permissions from the master account to those "sub-accounts" using the [x/authz module](#).

```

message GroupAccountInfo {

 // address is the group account address.
 string address = 1;

 // group_id is the ID of the Group the GroupAccount belongs to.
 uint64 group_id = 2;

 // admin is the account address of the group admin.
 string admin = 3;

 // metadata is any arbitrary metadata of this group account.
}

```

```

bytes metadata = 4;

// version is used to track changes to a group's GroupAccountInfo structure that
// invalidates active proposal from old versions.
uint64 version = 5;

// decision_policy specifies the group account's decision policy.
google.protobuf.Any decision_policy = 6 [(cosmos_proto.accepts_interface) =
"cosmos.group.v1.DecisionPolicy"];
}

```

Similarly to a group admin, a group account admin can update its metadata, decision policy or set a new group account admin.

A group account can also be an admin or a member of a group. For instance, a group admin could be another group account which could "elects" the members or it could be the same group that elects itself.

## Decision Policy

A decision policy is the mechanism by which members of a group can vote on proposals.

All decision policies should have a minimum and maximum voting window. The minimum voting window is the minimum duration that must pass in order for a proposal to potentially pass, and it may be set to 0. The maximum voting window is the maximum time that a proposal may be voted on and executed if it reached enough support before it is closed. Both of these values must be less than a chain-wide max voting window parameter.

We define the `DecisionPolicy` interface that all decision policies must implement:

```

type DecisionPolicy interface {
 codec.ProtoMarshaler

 ValidateBasic() error
 GetTimeout() types.Duration
 Allow(tally Tally, totalPower string, votingDuration time.Duration)
 (DecisionPolicyResult, error)
 Validate(g GroupInfo) error
}

type DecisionPolicyResult struct {
 Allow bool
 Final bool
}

```

### Threshold decision policy

A threshold decision policy defines a minimum support votes (yes), based on a tally of voter weights, for a proposal to pass. For this decision policy, abstain and veto are treated as no support (no).

```

message ThresholdDecisionPolicy {

 // threshold is the minimum weighted sum of support votes for a proposal to

```

```

succeed.

 string threshold = 1;

 // voting_period is the duration from submission of a proposal to the end of
 voting period
 // Within this period, votes and exec messages can be submitted.
 google.protobuf.Duration voting_period = 2 [(gogoproto.nullable) = false];
}

```

## Proposal

Any member of a group can submit a proposal for a group account to decide upon. A proposal consists of a set of `sdk.Msg`s that will be executed if the proposal passes as well as any metadata associated with the proposal. These `sdk.Msg`s get validated as part of the `Msg/CreateProposal` request validation. They should also have their signer set as the group account.

Internally, a proposal also tracks:

- its current `Status` : submitted, closed or aborted
- its `Result` : unfinalized, accepted or rejected
- its `VoteState` in the form of a `Tally`, which is calculated on new votes and when executing the proposal.

```

// Tally represents the sum of weighted votes.
message Tally {
 option (gogoproto.goproto_getters) = false;

 // yes_count is the weighted sum of yes votes.
 string yes_count = 1;

 // no_count is the weighted sum of no votes.
 string no_count = 2;

 // abstain_count is the weighted sum of abstainers.
 string abstain_count = 3;

 // veto_count is the weighted sum of vetoes.
 string veto_count = 4;
}

```

## Voting

Members of a group can vote on proposals. There are four choices to choose while voting - yes, no, abstain and veto. Not all decision policies will support them. Votes can contain some optional metadata. In the current implementation, the voting window begins as soon as a proposal is submitted.

Voting internally updates the proposal `VoteState` as well as `Status` and `Result` if needed.

## Executing Proposals

Proposals will not be automatically executed by the chain in this current design, but rather a user must submit a `Msg/Exec` transaction to attempt to execute the proposal based on the current votes and decision policy. A future upgrade could automate this and have the group account (or a fee granter) pay.

### Changing Group Membership

In the current implementation, updating a group or a group account after submitting a proposal will make it invalid. It will simply fail if someone calls `Msg/Exec` and will eventually be garbage collected.

### Notes on current implementation

This section outlines the current implementation used in the proof of concept of the group module but this could be subject to changes and iterated on.

#### ORM

The [ORM package](#) defines tables, sequences and secondary indexes which are used in the group module.

Groups are stored in state as part of a `groupTable`, the `group_id` being an auto-increment integer. Group members are stored in a `groupMemberTable`.

Group accounts are stored in a `groupAccountTable`. The group account address is generated based on an auto-increment integer which is used to derive the group module `RootModuleKey` into a `DerivedModuleKey`, as stated in [ADR-033](#). The group account is added as a new `ModuleAccount` through `x/auth`.

Proposals are stored as part of the `proposalTable` using the `Proposal` type. The `proposal_id` is an auto-increment integer.

Votes are stored in the `voteTable`. The primary key is based on the vote's `proposal_id` and `voter` account address.

#### ADR-033 to route proposal messages

Inter-module communication introduced by [ADR-033](#) can be used to route a proposal's messages using the `DerivedModuleKey` corresponding to the proposal's group account.

## Consequences

### Positive

- Improved UX for multi-signature accounts allowing key rotation and custom decision policies.

### Negative

### Neutral

- It uses ADR 033 so it will need to be implemented within the Cosmos SDK, but this doesn't imply necessarily any large refactoring of existing Cosmos SDK modules.
- The current implementation of the group module uses the ORM package.

## Further Discussions

- Convergence of `/group` and `x/gov` as both support proposals and voting:  
<https://github.com/cosmos/cosmos-sdk/discussions/9066>
- `x/group` possible future improvements:

- o Execute proposals on submission (<https://github.com/regen-network/regen-ledger/issues/288>)
- o Withdraw a proposal (<https://github.com/regen-network/cosmos-modules/issues/41>)
- o Make `Tally` more flexible and support non-binary choices

## References

- Initial specification:
  - o <https://gist.github.com/aaronc/b60628017352df5983791cad30babe56#group-module>
  - o [#5236](#)
- Proposal to add `x/group` into the Cosmos SDK: [#7633](#)

# ADR 43: NFT Module

## Changelog

- 2021-05-01: Initial Draft
- 2021-07-02: Review updates
- 2022-06-15: Add batch operation
- 2022-11-11: Remove strict validation of classID and tokenID

## Status

PROPOSED

## Abstract

This ADR defines the `x/nft` module which is a generic implementation of NFTs, roughly "compatible" with ERC721. **Applications using the `x/nft` module must implement the following functions:**

- `MsgNewClass` - Receive the user's request to create a class, and call the `NewClass` of the `x/nft` module.
- `MsgUpdateClass` - Receive the user's request to update a class, and call the `UpdateClass` of the `x/nft` module.
- `MsgMintNFT` - Receive the user's request to mint a nft, and call the `MintNFT` of the `x/nft` module.
- `BurnNFT` - Receive the user's request to burn a nft, and call the `BurnNFT` of the `x/nft` module.
- `UpdateNFT` - Receive the user's request to update a nft, and call the `UpdateNFT` of the `x/nft` module.

## Context

NFTs are more than just crypto art, which is very helpful for accruing value to the Cosmos ecosystem. As a result, Cosmos Hub should implement NFT functions and enable a unified mechanism for storing and sending the ownership representative of NFTs as discussed in <https://github.com/cosmos/cosmos-sdk/discussions/9065>.

As discussed in [#9065](#), several potential solutions can be considered:

- irismod/nft and modules/incubator/nft
- CW721

- DID NFTs
- interNFT

Since functions/use cases of NFTs are tightly connected with their logic, it is almost impossible to support all the NFTs' use cases in one Cosmos SDK module by defining and implementing different transaction types.

Considering generic usage and compatibility of interchain protocols including IBC and Gravity Bridge, it is preferred to have a generic NFT module design which handles the generic NFTs logic. This design idea can enable composability that application-specific functions should be managed by other modules on Cosmos Hub or on other Zones by importing the NFT module.

The current design is based on the work done by [IRISnet team](#) and an older implementation in the [Cosmos repository](#).

## Decision

We create a `x/nft` module, which contains the following functionality:

- Store NFTs and track their ownership.
- Expose `Keeper` interface for composing modules to transfer, mint and burn NFTs.
- Expose external `Message` interface for users to transfer ownership of their NFTs.
- Query NFTs and their supply information.

The proposed module is a base module for NFT app logic. It's goal is to provide a common layer for storage, basic transfer functionality and IBC. The module should not be used as a standalone. Instead an app should create a specialized module to handle app specific logic (eg: NFT ID construction, royalty), user level minting and burning. Moreover an app specialized module should handle auxiliary data to support the app logic (eg indexes, ORM, business data).

All data carried over IBC must be part of the `NFT` or `Class` type described below. The app specific NFT data should be encoded in `NFT.data` for cross-chain integrity. Other objects related to NFT, which are not important for integrity can be part of the app specific module.

## Types

We propose two main types:

- `Class` -- describes NFT class. We can think about it as a smart contract address.
- `NFT` -- object representing unique, non fungible asset. Each NFT is associated with a Class.

### Class

**NFT Class** is comparable to an ERC-721 smart contract (provides description of a smart contract), under which a collection of NFTs can be created and managed.

```
message Class {
 string id = 1;
 string name = 2;
 string symbol = 3;
 string description = 4;
 string uri = 5;
 string uri_hash = 6;
```

```
 google.protobuf.Any data = 7;
}
```

- `id` is used as the primary index for storing the class; *required*
- `name` is a descriptive name of the NFT class; *optional*
- `symbol` is the symbol usually shown on exchanges for the NFT class; *optional*
- `description` is a detailed description of the NFT class; *optional*
- `uri` is a URI for the class metadata stored off chain. It should be a JSON file that contains metadata about the NFT class and NFT data schema ([OpenSea example](#)); *optional*
- `uri_hash` is a hash of the document pointed by `uri`; *optional*
- `data` is app specific metadata of the class; *optional*

## NFT

We define a general model for `NFT` as follows.

```
message NFT {
 string class_id = 1;
 string id = 2;
 string uri = 3;
 string uri_hash = 4;
 google.protobuf.Any data = 10;
}
```

- `class_id` is the identifier of the NFT class where the NFT belongs; *required*
- `id` is an identifier of the NFT, unique within the scope of its class. It is specified by the creator of the NFT and may be expanded to use DID in the future. `class_id` combined with `id` uniquely identifies an NFT and is used as the primary index for storing the NFT; *required*

```
{class_id}/{id} --> NFT (bytes)
```

- `uri` is a URI for the NFT metadata stored off chain. Should point to a JSON file that contains metadata about this NFT (Ref: [ERC721 standard and OpenSea extension](#)); *required*
- `uri_hash` is a hash of the document pointed by `uri`; *optional*
- `data` is an app specific data of the NFT. CAN be used by composing modules to specify additional properties of the NFT; *optional*

This ADR doesn't specify values that `data` can take; however, best practices recommend upper-level NFT modules clearly specify their contents. Although the value of this field doesn't provide the additional context required to manage NFT records, which means that the field can technically be removed from the specification, the field's existence allows basic informational/UI functionality.

## Keeper Interface

```
type Keeper interface {
 NewClass(ctx sdk.Context, class Class)
 UpdateClass(ctx sdk.Context, class Class)
```

```

Mint(ctx sdk.Context, nft NFT, receiver sdk.AccAddress) // updates totalSupply
BatchMint(ctx sdk.Context, tokens []NFT, receiver sdk.AccAddress) error

Burn(ctx sdk.Context, classId string, nftId string) // updates totalSupply
BatchBurn(ctx sdk.Context, classID string, nftIDs []string) error

Update(ctx sdk.Context, nft NFT)
BatchUpdate(ctx sdk.Context, tokens []NFT) error

Transfer(ctx sdk.Context, classId string, nftId string, receiver sdk.AccAddress)
BatchTransfer(ctx sdk.Context, classID string, nftIDs []string, receiver
sdk.AccAddress) error

GetClass(ctx sdk.Context, classId string) Class
GetClasses(ctx sdk.Context) []Class

GetNFT(ctx sdk.Context, classId string, nftId string) NFT
GetNFTsOfClassByOwner(ctx sdk.Context, classId string, owner sdk.AccAddress) []NFT
GetNFTsOfClass(ctx sdk.Context, classId string) []NFT

GetOwner(ctx sdk.Context, classId string, nftId string) sdk.AccAddress
GetBalance(ctx sdk.Context, classId string, owner sdk.AccAddress) uint64
GetTotalSupply(ctx sdk.Context, classId string) uint64
}

```

Other business logic implementations should be defined in composing modules that import `x/nft` and use its `Keeper`.

## Msg Service

```

service Msg {
 rpc Send(MsgSend) returns (MsgSendResponse);
}

message MsgSend {
 string class_id = 1;
 string id = 2;
 string sender = 3;
 string receiver = 4;
}
message MsgSendResponse {}

```

`MsgSend` can be used to transfer the ownership of an NFT to another address.

The implementation outline of the server is as follows:

```

type msgServer struct{
 k Keeper
}

```

```

func (m msgServer) Send(ctx context.Context, msg *types.MsgSend)
(*types.MsgSendResponse, error) {
 // check current ownership
 assertEquals(msg.Sender, m.k.GetOwner(msg.ClassId, msg.Id))

 // transfer ownership
 m.k.Transfer(msg.ClassId, msg.Id, msg.Receiver)

 return &types.MsgSendResponse{}, nil
}

```

The query service methods for the `x/nft` module are:

```

service Query {
 // Balance queries the number of NFTs of a given class owned by the owner, same as
 balanceOf in ERC721
 rpc Balance(QueryBalanceRequest) returns (QueryBalanceResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/balance/{owner}/{class_id}";
 }

 // Owner queries the owner of the NFT based on its class and id, same as ownerOf
 in ERC721
 rpc Owner(QueryOwnerRequest) returns (QueryOwnerResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/owner/{class_id}/{id}";
 }

 // Supply queries the number of NFTs from the given class, same as totalSupply of
 ERC721.
 rpc Supply(QuerySupplyRequest) returns (QuerySupplyResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/supply/{class_id}";
 }

 // NFTs queries all NFTs of a given class or owner, choose at least one of the two,
 similar to tokenByIndex in ERC721Enumerable
 rpc NFTs(QueryNFTsRequest) returns (QueryNFTsResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/nfts";
 }

 // NFT queries an NFT based on its class and id.
 rpc NFT(QueryNFTRequest) returns (QueryNFTResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/nfts/{class_id}/{id}";
 }

 // Class queries an NFT class based on its id
 rpc Class(QueryClassRequest) returns (QueryClassResponse) {
 option (google.api.http).get = "/cosmos/nft/v1beta1/classes/{class_id}";
 }

 // Classes queries all NFT classes
 rpc Classes(QueryClassesRequest) returns (QueryClassesResponse) {

```

```

 option (google.api.http).get = "/cosmos/nft/v1beta1/classes";
}

}

// QueryBalanceRequest is the request type for the Query/Balance RPC method
message QueryBalanceRequest {
 string class_id = 1;
 string owner = 2;
}

// QueryBalanceResponse is the response type for the Query/Balance RPC method
message QueryBalanceResponse {
 uint64 amount = 1;
}

// QueryOwnerRequest is the request type for the Query/Owner RPC method
message QueryOwnerRequest {
 string class_id = 1;
 string id = 2;
}

// QueryOwnerResponse is the response type for the Query/Owner RPC method
message QueryOwnerResponse {
 string owner = 1;
}

// QuerySupplyRequest is the request type for the Query/Supply RPC method
message QuerySupplyRequest {
 string class_id = 1;
}

// QuerySupplyResponse is the response type for the Query/Supply RPC method
message QuerySupplyResponse {
 uint64 amount = 1;
}

// QueryNFTstRequest is the request type for the Query/NFTs RPC method
message QueryNFTsRequest {
 string class_id = 1;
 string owner = 2;
 cosmos.base.query.v1beta1.PageRequest pagination = 3;
}

// QueryNFTsResponse is the response type for the Query/NFTs RPC methods
message QueryNFTsResponse {
 repeated cosmos.nft.v1beta1.NFT nfts = 1;
 cosmos.base.query.v1beta1.PageResponse pagination = 2;
}

// QueryNFTRequest is the request type for the Query/NFT RPC method
message QueryNFTRequest {
 string class_id = 1;
}

```

```

 string id = 2;
}

// QueryNFTResponse is the response type for the Query/NFT RPC method
message QueryNFTResponse {
 cosmos.nft.v1beta1.NFT nft = 1;
}

// QueryClassRequest is the request type for the Query/Class RPC method
message QueryClassRequest {
 string class_id = 1;
}

// QueryClassResponse is the response type for the Query/Class RPC method
message QueryClassResponse {
 cosmos.nft.v1beta1.Class class = 1;
}

// QueryClassesRequest is the request type for the Query/Classes RPC method
message QueryClassesRequest {
 // pagination defines an optional pagination for the request.
 cosmos.base.query.v1beta1.PageRequest pagination = 1;
}

// QueryClassesResponse is the response type for the Query/Classes RPC method
message QueryClassesResponse {
 repeated cosmos.nft.v1beta1.Class classes = 1;
 cosmos.base.query.v1beta1.PageResponse pagination = 2;
}

```

## Interoperability

Interoperability is all about reusing assets between modules and chains. The former one is achieved by ADR-33: Protobuf client - server communication. At the time of writing ADR-33 is not finalized. The latter is achieved by IBC. Here we will focus on the IBC side. IBC is implemented per module. Here, we aligned that NFTs will be recorded and managed in the x/nft. This requires creation of a new IBC standard and implementation of it.

For IBC interoperability, NFT custom modules MUST use the NFT object type understood by the IBC client. So, for x/nft interoperability, custom NFT implementations (example: x/cryptokitty) should use the canonical x/nft module and proxy all NFT balance keeping functionality to x/nft or else re-implement all functionality using the NFT object type understood by the IBC client. In other words: x/nft becomes the standard NFT registry for all Cosmos NFTs (example: x/cryptokitty will register a kitty NFT in x/nft and use x/nft for book keeping). This was [discussed](#) in the context of using x/bank as a general asset balance book. Not using x/nft will require implementing another module for IBC.

## Consequences

### Backward Compatibility

No backward incompatibilities.

## Forward Compatibility

This specification conforms to the ERC-721 smart contract specification for NFT identifiers. Note that ERC-721 defines uniqueness based on (contract address, uint256 tokenId), and we conform to this implicitly because a single module is currently aimed to track NFT identifiers. Note: use of the (mutable) data field to determine uniqueness is not safe.

## Positive

- NFT identifiers available on Cosmos Hub.
- Ability to build different NFT modules for the Cosmos Hub, e.g., ERC-721.
- NFT module which supports interoperability with IBC and other cross-chain infrastructures like Gravity Bridge

## Negative

- New IBC app is required for x/nft
- CW721 adapter is required

## Neutral

- Other functions need more modules. For example, a custody module is needed for NFT trading function, a collectible module is needed for defining NFT properties.

## Further Discussions

For other kinds of applications on the Hub, more app-specific modules can be developed in the future:

- `x/nft/custody` : custody of NFTs to support trading functionality.
- `x/nft/marketplace` : selling and buying NFTs using sdk.Coins.
- `x/fractional` : a module to split an ownership of an asset (NFT or other assets) for multiple stakeholder. `x/group` should work for most of the cases.

Other networks in the Cosmos ecosystem could design and implement their own NFT modules for specific NFT applications and use cases.

## References

- Initial discussion: <https://github.com/cosmos/cosmos-sdk/discussions/9065>
- x/nft: initialize module: <https://github.com/cosmos/cosmos-sdk/pull/9174>
- [ADR 033](#)

# ADR 044: Guidelines for Updating Protobuf

## Definitions

### Changelog

- 28.06.2021: Initial Draft
- 02.12.2021: Add `Since:` comment for new fields
- 21.07.2022: Remove the rule of no new `Msg` in the same proto version.

### Status

Draft

## Abstract

This ADR provides guidelines and recommended practices when updating Protobuf definitions. These guidelines are targeting module developers.

## Context

The Cosmos SDK maintains a set of [Protobuf definitions](#). It is important to correctly design Protobuf definitions to avoid any breaking changes within the same version. The reasons are to not break tooling (including indexers and explorers), wallets and other third-party integrations.

When making changes to these Protobuf definitions, the Cosmos SDK currently only follows [Buf's](#) recommendations. We noticed however that Buf's recommendations might still result in breaking changes in the SDK in some cases. For example:

- Adding fields to `Msg`s. Adding fields is a not a Protobuf spec-breaking operation. However, when adding new fields to `Msg`s, the unknown field rejection will throw an error when sending the new `Msg` to an older node.
- Marking fields as `reserved`. Protobuf proposes the `reserved` keyword for removing fields without the need to bump the package version. However, by doing so, client backwards compatibility is broken as Protobuf doesn't generate anything for `reserved` fields. See [#9446](#) for more details on this issue.

Moreover, module developers often face other questions around Protobuf definitions such as "Can I rename a field?" or "Can I deprecate a field?" This ADR aims to answer all these questions by providing clear guidelines about allowed updates for Protobuf definitions.

## Decision

We decide to keep [Buf's](#) recommendations with the following exceptions:

- `UNARY_RPC` : the Cosmos SDK currently does not support streaming RPCs.
- `COMMENT_FIELD` : the Cosmos SDK allows fields with no comments.
- `SERVICE_SUFFIX` : we use the `Query` and `Msg` service naming convention, which doesn't use the `-Service` suffix.
- `PACKAGE_VERSION_SUFFIX` : some packages, such as `cosmos.crypto.ed25519`, don't use a version suffix.
- `RPC_REQUEST_STANDARD_NAME` : Requests for the `Msg` service don't have the `-Request` suffix to keep backwards compatibility.

On top of Buf's recommendations we add the following guidelines that are specific to the Cosmos SDK.

### Updating Protobuf Definition Without Bumping Version

#### 1. Module developers MAY add new Protobuf definitions

Module developers MAY add new `message`s, new `Service`s, new `rpc` endpoints, and new fields to existing messages. This recommendation follows the Protobuf specification, but is added in this document for clarity, as the SDK requires one additional change.

The SDK requires the Protobuf comment of the new addition to contain one line with the following format:

```
// Since: cosmos-sdk <version>{}, <version>...}
```

Where each `version` denotes a minor ("0.45") or patch ("0.44.5") version from which the field is available. This will greatly help client libraries, who can optionally use reflection or custom code generation to show/hide these fields depending on the targeted node version.

As examples, the following comments are valid:

```
// Since: cosmos-sdk 0.44

// Since: cosmos-sdk 0.42.11, 0.44.5
```

and the following ones are NOT valid:

```
// Since cosmos-sdk v0.44

// since: cosmos-sdk 0.44

// Since: cosmos-sdk 0.42.11 0.44.5

// Since: Cosmos SDK 0.42.11, 0.44.5
```

## 2. Fields MAY be marked as `deprecated`, and nodes MAY implement a protocol-breaking change for handling these fields

Protobuf supports the [deprecated field option](#), and this option MAY be used on any field, including `Msg` fields. If a node handles a Protobuf message with a non-empty deprecated field, the node MAY change its behavior upon processing it, even in a protocol-breaking way. When possible, the node MUST handle backwards compatibility without breaking the consensus (unless we increment the proto version).

As an example, the Cosmos SDK v0.42 to v0.43 update contained two Protobuf-breaking changes, listed below. Instead of bumping the package versions from `v1beta1` to `v1`, the SDK team decided to follow this guideline, by reverting the breaking changes, marking those changes as deprecated, and modifying the node implementation when processing messages with deprecated fields. More specifically:

- The Cosmos SDK recently removed support for [time-based software upgrades](#). As such, the `time` field has been marked as deprecated in `cosmos.upgrade.v1beta1.Plan`. Moreover, the node will reject any proposal containing an upgrade Plan whose `time` field is non-empty.
- The Cosmos SDK now supports [governance split votes](#). When querying for votes, the returned `cosmos.gov.v1beta1.Vote` message has its `option` field (used for 1 vote option) deprecated in favor of its `options` field (allowing multiple vote options). Whenever possible, the SDK still populates the deprecated `option` field, that is, if and only if the `len(options) == 1` and `options[0].Weight == 1.0`.

## 3. Fields MUST NOT be renamed

Whereas the official Protobuf recommendations do not prohibit renaming fields, as it does not break the Protobuf binary representation, the SDK explicitly forbids renaming fields in Protobuf structs. The main reason for this choice is to avoid introducing breaking changes for clients, which often rely on hard-coded fields from generated types. Moreover, renaming fields will lead to client-breaking JSON representations of Protobuf definitions, used in REST endpoints and in the CLI.

## Incrementing Protobuf Package Version

TODO, needs architecture review. Some topics:

- Bumping versions frequency
- When bumping versions, should the Cosmos SDK support both versions?
  - i.e. v1beta1 -> v1, should we have two folders in the Cosmos SDK, and handlers for both versions?
- mention ADR-023 Protobuf naming

## Consequences

*This section describes the resulting context, after applying the decision. All consequences should be listed here, not just the "positive" ones. A particular decision may have positive, negative, and neutral consequences, but all of them affect the team and project in the future.*

### Backwards Compatibility

*All ADRs that introduce backwards incompatibilities must include a section describing these incompatibilities and their severity. The ADR must explain how the author proposes to deal with these incompatibilities. ADR submissions without a sufficient backwards compatibility treatise may be rejected outright.*

#### Positive

- less pain to tool developers
- more compatibility in the ecosystem
- ...

#### Negative

{negative consequences}

#### Neutral

- more rigor in Protobuf review

## Further Discussions

This ADR is still in the DRAFT stage, and the "Incrementing Protobuf Package Version" will be filled in once we make a decision on how to correctly do it.

## Test Cases [optional]

Test cases for an implementation are mandatory for ADRs that are affecting consensus changes. Other ADRs can choose to include links to test cases if applicable.

## References

- [#9445](#) Release proto definitions v1
- [#9446](#) Address v1beta1 proto breaking changes

# ADR 045: BaseApp {Check, Deliver}Tx as Middlewares

## Changelog

- 20.08.2021: Initial draft.
- 07.12.2021: Update `tx.Handler` interface ([#10693](#)).
- 17.05.2022: ADR is abandoned, as middlewares are deemed too hard to reason about.

## Status

ABANDONED. Replacement is being discussed in [#11955](#).

## Abstract

This ADR replaces the current BaseApp `runTx` and antehandlers design with a middleware-based design.

## Context

BaseApp's implementation of ABCI `{Check, Deliver}Tx()` and its own `Simulate()` method call the `runTx` method under the hood, which first runs antehandlers, then executes `Msg`s. However, the [transaction Tips](#) and [refunding unused gas](#) use cases require custom logic to be run after the `Msg`s execution. There is currently no way to achieve this.

An naive solution would be to add post- `Msg` hooks to BaseApp. However, the Cosmos SDK team thinks in parallel about the bigger picture of making app wiring simpler ([#9181](#)), which includes making BaseApp more lightweight and modular.

## Decision

We decide to transform Baseapp's implementation of ABCI `{Check, Deliver}Tx` and its own `Simulate` methods to use a middleware-based design.

The two following interfaces are the base of the middleware design, and are defined in `types/tx`:

```
type Handler interface {
 CheckTx(ctx context.Context, req Request, checkReq RequestCheckTx) (Response,
 ResponseCheckTx, error)
 DeliverTx(ctx context.Context, req Request) (Response, error)
 SimulateTx(ctx context.Context, req Request) (Response, error)
}

type Middleware func(Handler) Handler
```

where we define the following arguments and return types:

```
type Request struct {
 Tx sdk.Tx
```

```

 TxBytes []byte
}

type Response struct {
 GasWanted uint64
 GasUsed uint64
 // MsgResponses is an array containing each Msg service handler's response
 // type, packed in an Any. This will get proto-serialized into the `Data` field
 // in the ABCI Check/DeliverTx responses.
 MsgResponses []*codectypes.Any
 Log string
 Events []abci.Event
}

type RequestCheckTx struct {
 Type abci.CheckTxType
}

type ResponseCheckTx struct {
 Priority int64
}

```

Please note that because CheckTx handles separate logic related to mempool prioritization, its signature is different than DeliverTx and SimulateTx.

BaseApp holds a reference to a `tx.Handler`:

```

type BaseApp struct {
 // other fields
 txHandler tx.Handler
}

```

Baseapp's ABCI `{Check, Deliver}Tx()` and `Simulate()` methods simply call `app.txHandler.{Check, Deliver, Simulate}Tx()` with the relevant arguments. For example, for `DeliverTx`:

```

func (app *BaseApp) DeliverTx(req abci.RequestDeliverTx) abci.ResponseDeliverTx {
 var abciRes abci.ResponseDeliverTx
 ctx := app.getContextForTx(runTxModeDeliver, req.Tx)
 res, err := app.txHandler.DeliverTx(ctx, tx.Request{TxBytes: req.Tx})
 if err != nil {
 abciRes = sdkerrors.ResponseDeliverTx(err, uint64(res.GasUsed),
 uint64(res.GasWanted), app.trace)
 return abciRes
 }

 abciRes, err = convertTxResponseToDeliverTx(res)
 if err != nil {
 return sdkerrors.ResponseDeliverTx(err, uint64(res.GasUsed),
 uint64(res.GasWanted), app.trace)
 }
}

```

```

 return abciRes
}

// convertTxResponseToDeliverTx converts a tx.Response into a
abci.ResponseDeliverTx.
func convertTxResponseToDeliverTx(txRes tx.Response) (abci.ResponseDeliverTx, error)
{
 data, err := makeABCIData(txRes)
 if err != nil {
 return abci.ResponseDeliverTx{}, nil
 }

 return abci.ResponseDeliverTx{
 Data: data,
 Log: txRes.Log,
 Events: txRes.Events,
 }, nil
}

// makeABCIData generates the Data field to be sent to ABCI Check/DeliverTx.
func makeABCIData(txRes tx.Response) ([]byte, error) {
 return proto.Marshal(&sdk.TxMsgData{MsgResponses: txRes.MsgResponses})
}

```

The implementations are similar for `BaseApp.CheckTx` and `BaseApp.Simulate`.

`baseapp.txHandler`'s three methods' implementations can obviously be monolithic functions, but for modularity we propose a middleware composition design, where a middleware is simply a function that takes a `tx.Handler`, and returns another `tx.Handler` wrapped around the previous one.

## Implementing a Middleware

In practice, middlewares are created by Go function that takes as arguments some parameters needed for the middleware, and returns a `tx.Middleware`.

For example, for creating an arbitrary `MyMiddleware`, we can implement:

```

// myTxHandler is the tx.Handler of this middleware. Note that it holds a
// reference to the next tx.Handler in the stack.
type myTxHandler struct {
 // next is the next tx.Handler in the middleware stack.
 next tx.Handler
 // some other fields that are relevant to the middleware can be added here
}

// NewMyMiddleware returns a middleware that does this and that.
func NewMyMiddleware(arg1, arg2) tx.Middleware {
 return func (txh tx.Handler) tx.Handler {
 return myTxHandler{
 next: txh,
 // optionally, set arg1, arg2... if they are needed in the middleware
 }
 }
}

```

```

 }

 }

// Assert myTxHandler is a tx.Handler.
var _ tx.Handler = myTxHandler{}

func (h myTxHandler) CheckTx(ctx context.Context, req Request, checkReq
RequestCheckTx) (Response, ResponseCheckTx, error) {
 // CheckTx specific pre-processing logic

 // run the next middleware
 res, checkRes, err := txh.next.CheckTx(ctx, req, checkReq)

 // CheckTx specific post-processing logic

 return res, checkRes, err
}

func (h myTxHandler) DeliverTx(ctx context.Context, req Request) (Response, error) {
 // DeliverTx specific pre-processing logic

 // run the next middleware
 res, err := txh.next.DeliverTx(ctx, tx, req)

 // DeliverTx specific post-processing logic

 return res, err
}

func (h myTxHandler) SimulateTx(ctx context.Context, req Request) (Response, error) {
 // SimulateTx specific pre-processing logic

 // run the next middleware
 res, err := txh.next.SimulateTx(ctx, tx, req)

 // SimulateTx specific post-processing logic

 return res, err
}

```

## Composing Middlewares

While BaseApp simply holds a reference to a `tx.Handler`, this `tx.Handler` itself is defined using a middleware stack. The Cosmos SDK exposes a base (i.e. innermost) `tx.Handler` called `RunMsgsTxHandler`, which executes messages.

Then, the app developer can compose multiple middlewares on top on the base `tx.Handler`. Each middleware can run pre-and-post-processing logic around its next middleware, as described in the section above. Conceptually, as an example, given the middlewares `A`, `B`, and `C` and the base `tx.Handler` `H` the stack looks like:

```

A.pre
B.pre
C.pre
 H # The base tx.handler, for example `RunMsgsTxHandler`
C.post
B.post
A.post

```

We define a `ComposeMiddlewares` function for composing middlewares. It takes the base handler as first argument, and middlewares in the "outer to inner" order. For the above stack, the final `tx.Handler` is:

```
txHandler := middleware.ComposeMiddlewares(H, A, B, C)
```

The middleware is set in `BaseApp` via its `SetTxHandler` setter:

```

// simapp/app.go

txHandler := middleware.ComposeMiddlewares(...)
app.SetTxHandler(txHandler)

```

The app developer can define their own middlewares, or use the Cosmos SDK's pre-defined middlewares from `middleware.NewDefaultTxHandler()`.

## Middlewares Maintained by the Cosmos SDK

While the app developer can define and compose the middlewares of their choice, the Cosmos SDK provides a set of middlewares that caters for the ecosystem's most common use cases. These middlewares are:

| Middleware              | Description                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RunMsgsTxHandler        | This is the base <code>tx.Handler</code> . It replaces the old <code>baseapp</code> 's <code>runMsgs</code> , and executes a transaction's <code>Msgs</code> .                                                                                                                                                                                                      |
| TxDecoderMiddleware     | This middleware takes in transaction raw bytes, and decodes them into a <code>sdk.Tx</code> . It replaces the <code>baseapp.txDecoder</code> field, so that <code>BaseApp</code> stays as thin as possible. Since most middlewares read the contents of the <code>sdk.Tx</code> , the <code>TxDecoderMiddleware</code> should be run first in the middleware stack. |
| {Antehandlers}          | Each antehandler is converted to its own middleware. These middlewares perform signature verification, fee deductions and other validations on the incoming transaction.                                                                                                                                                                                            |
| IndexEventsTxMiddleware | This is a simple middleware that chooses which events to index in Tendermint. Replaces <code>baseapp.indexEvents</code> (which unfortunately still exists in <code>baseapp</code> too, because it's used to index Begin/EndBlock events)                                                                                                                            |
| RecoveryTxMiddleware    | This index recovers from panics. It replaces <code>baseapp.runTx</code> 's panic recovery described in <a href="#">ADR-022</a> .                                                                                                                                                                                                                                    |

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GasTxMiddleware | This replaces the <a href="#">Setup</a> Antehandler. It sets a GasMeter on sdk.Context. Note that before, GasMeter was set on sdk.Context inside the antehandlers, and there was some mess around the fact that antehandlers had their own panic recovery system so that the GasMeter could be read by baseapp's recovery system. Now, this mess is all removed: one middleware sets GasMeter, another one handles recovery. |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Similarities and Differences between Antehandlers and Middlewares

The middleware-based design builds upon the existing antehandlers design described in [ADR-010](#). Even though the final decision of ADR-010 was to go with the "Simple Decorators" approach, the middleware design is actually very similar to the other [Decorator Pattern](#) proposal, also used in [weave](#).

### Similarities with Antehandlers

- Designed as chaining/composing small modular pieces.
- Allow code reuse for `{Check, Deliver}Tx` and for `Simulate`.
- Set up in `app.go`, and easily customizable by app developers.
- Order is important.

### Differences with Antehandlers

- The Antehandlers are run before `Msg` execution, whereas middlewares can run before and after.
- The middleware approach uses separate methods for `{Check, Deliver, Simulate}Tx`, whereas the antehandlers pass a `simulate bool` flag and uses the `sdkCtx.Is{Check, Recheck}Tx()` flags to determine in which transaction mode we are.
- The middleware design lets each middleware hold a reference to the next middleware, whereas the antehandlers pass a `next` argument in the `AnteHandle` method.
- The middleware design use Go's standard `context.Context`, whereas the antehandlers use `sdk.Context`.

## Consequences

### Backwards Compatibility

Since this refactor removes some logic away from BaseApp and into middlewares, it introduces API-breaking changes for app developers. Most notably, instead of creating an antehandler chain in `app.go`, app developers need to create a middleware stack:

```

- anteHandler, err := ante.NewAnteHandler(
- ante.HandlerOptions{
- AccountKeeper: app.AccountKeeper,
- BankKeeper: app.BankKeeper,
- SignModeHandler: encodingConfig.TxConfig.SignModeHandler(),
- FeegrantKeeper: app.FeeGrantKeeper,
- SigGasConsumer: ante.DefaultSigVerificationGasConsumer,
- },
-)
+txHandler, err :=
authmiddleware.NewDefaultTxHandler(authmiddleware.TxHandlerOptions{
+ Debug: app.Trace(),
+ IndexEvents: indexEvents,

```

```

+ LegacyRouter: app.legacyRouter,
+ MsgServiceRouter: app.msgSvcRouter,
+ LegacyAnteHandler: anteHandler,
+ TxDecoder: encodingConfig.TxConfig.TxDecoder,
+ })
if err != nil {
 panic(err)
}
- app.SetAnteHandler(anteHandler)
+ app.SetTxHandler(txHandler)

```

Other more minor API breaking changes will also be provided in the CHANGELOG. As usual, the Cosmos SDK will provide a release migration document for app developers.

This ADR does not introduce any state-machine-, client- or CLI-breaking changes.

## Positive

- Allow custom logic to be run before and after `Msg` execution. This enables the [tips](#) and [gas refund](#) use cases, and possibly other ones.
- Make BaseApp more lightweight, and defer complex logic to small modular components.
- Separate paths for `{Check, Deliver, Simulate}Tx` with different return types. This allows for improved readability (replace `if sdkCtx.IsRecheckTx() && !simulate {...}` with separate methods) and more flexibility (e.g. returning a `priority` in `ResponseCheckTx`).

## Negative

- It is hard to understand at first glance the state updates that would occur after a middleware runs given the `sdk.Context` and `tx`. A middleware can have an arbitrary number of nested middleware being called within its function body, each possibly doing some pre- and post-processing before calling the next middleware on the chain. Thus to understand what a middleware is doing, one must also understand what every other middleware further along the chain is also doing, and the order of middlewares matters. This can get quite complicated to understand.
- API-breaking changes for app developers.

## Neutral

No neutral consequences.

## Further Discussions

- [#9934](#) Decomposing BaseApp's other ABCI methods into middlewares.
- Replace `sdk.Tx` interface with the concrete protobuf Tx type in the `tx.Handler` methods signature.

## Test Cases

We update the existing baseapp and antehandlers tests to use the new middleware API, but keep the same test cases and logic, to avoid introducing regressions. Existing CLI tests will also be left untouched.

For new middlewares, we introduce unit tests. Since middlewares are purposefully small, unit tests suit well.

## References

- Initial discussion: <https://github.com/cosmos/cosmos-sdk/issues/9585>
- Implementation: [#9920 BaseApp refactor](#) and [#10028 Antehandlers migration](#)

# ADR 046: Module Params

## Changelog

- Sep 22, 2021: Initial Draft

## Status

ACCEPTED

## Abstract

This ADR describes an alternative approach to how Cosmos SDK modules use, interact, and store their respective parameters.

## Context

Currently, in the Cosmos SDK, modules that require the use of parameters use the `x/params` module. The `x/params` works by having modules define parameters, typically via a simple `Params` structure, and registering that structure in the `x/params` module via a unique `Subspace` that belongs to the respective registering module. The registering module then has unique access to its respective `Subspace`. Through this `Subspace`, the module can get and set its `Params` structure.

In addition, the Cosmos SDK's `x/gov` module has direct support for changing parameters on-chain via a `ParamChangeProposal` governance proposal type, where stakeholders can vote on suggested parameter changes.

There are various tradeoffs to using the `x/params` module to manage individual module parameters. Namely, managing parameters essentially comes for "free" in that developers only need to define the `Params` struct, the `Subspace`, and the various auxiliary functions, e.g. `ParamSetPairs`, on the `Params` type. However, there are some notable drawbacks. These drawbacks include the fact that parameters are serialized in state via JSON which is extremely slow. In addition, parameter changes via `ParamChangeProposal` governance proposals have no way of reading from or writing to state. In other words, it is currently not possible to have any state transitions in the application during an attempt to change param(s).

## Decision

We will build off of the alignment of `x/gov` and `x/authz` work per [#9810](#). Namely, module developers will create one or more unique parameter data structures that must be serialized to state. The Param data structures must implement `sdk.Msg` interface with respective Protobuf Msg service method which will validate and update the parameters with all necessary changes. The `x/gov` module via the work done in [#9810](#), will dispatch Param messages, which will be handled by Protobuf Msg services.

Note, it is up to developers to decide how to structure their parameters and the respective `sdk.Msg` messages. Consider the parameters currently defined in `x/auth` using the `x/params` module for parameter management:

```

message Params {
 uint64 max_memo_characters = 1;
 uint64 tx_sig_limit = 2;
 uint64 tx_size_cost_per_byte = 3;
 uint64 sig_verify_cost_ed25519 = 4;
 uint64 sig_verify_cost_secp256k1 = 5;
}

```

Developers can choose to either create a unique data structure for every field in `Params` or they can create a single `Params` structure as outlined above in the case of `x/auth`.

In the former, `x/params`, approach, a `sdk.Msg` would need to be created for every single field along with a handler. This can become burdensome if there are a lot of parameter fields. In the latter case, there is only a single data structure and thus only a single message handler, however, the message handler might have to be more sophisticated in that it might need to understand what parameters are being changed vs what parameters are untouched.

Params change proposals are made using the `x/gov` module. Execution is done through `x/authz` authorization to the root `x/gov` module's account.

Continuing to use `x/auth`, we demonstrate a more complete example:

```

type Params struct {
 MaxMemoCharacters uint64
 TxSigLimit uint64
 TxSizeCostPerByte uint64
 SigVerifyCostED25519 uint64
 SigVerifyCostSecp256k1 uint64
}

type MsgUpdateParams struct {
 MaxMemoCharacters uint64
 TxSigLimit uint64
 TxSizeCostPerByte uint64
 SigVerifyCostED25519 uint64
 SigVerifyCostSecp256k1 uint64
}

type MsgUpdateParamsResponse struct {}

func (ms msgServer) UpdateParams(goCtx context.Context, msg *types.MsgUpdateParams)
(*types.MsgUpdateParamsResponse, error) {
 ctx := sdk.UnwrapSDKContext(goCtx)

 // verification logic...

 // persist params
 params := ParamsFromMsg(msg)
 ms.SaveParams(ctx, params)

 return &types.MsgUpdateParamsResponse{}, nil
}

```

```

}

func ParamsFromMsg(msg *types.MsgUpdateParams) Params {
 // ...
}

```

A gRPC `Service` query should also be provided, for example:

```

service Query {
 // ...

 rpc Params(QueryParamsRequest) returns (QueryParamsResponse) {
 option (google.api.http).get = "/cosmos/<module>/v1beta1/params";
 }
}

message QueryParamsResponse {
 Params params = 1 [(gogoproto.nullable) = false];
}

```

## Consequences

As a result of implementing the module parameter methodology, we gain the ability for module parameter changes to be stateful and extensible to fit nearly every application's use case. We will be able to emit events (and trigger hooks registered to that events using the work proposed in [event hooks](#)), call other Msg service methods or perform migration. In addition, there will be significant gains in performance when it comes to reading and writing parameters from and to state, especially if a specific set of parameters are read on a consistent basis.

However, this methodology will require developers to implement more types and Msg service methods which can become burdensome if many parameters exist. In addition, developers are required to implement persistence logics of module parameters. However, this should be trivial.

## Backwards Compatibility

The new method for working with module parameters is naturally not backwards compatible with the existing `x/params` module. However, the `x/params` will remain in the Cosmos SDK and will be marked as deprecated with no additional functionality being added apart from potential bug fixes. Note, the `x/params` module may be removed entirely in a future release.

## Positive

- Module parameters are serialized more efficiently
- Modules are able to react on parameters changes and perform additional actions.
- Special events can be emitted, allowing hooks to be triggered.

## Negative

- Module parameters becomes slightly more burdensome for module developers:
  - Modules are now responsible for persisting and retrieving parameter state

- Modules are now required to have unique message handlers to handle parameter changes per unique parameter data structure.

## Neutral

- Requires [#9810](#) to be reviewed and merged.

## References

- <https://github.com/cosmos/cosmos-sdk/pull/9810>
- <https://github.com/cosmos/cosmos-sdk/issues/9438>
- <https://github.com/cosmos/cosmos-sdk/discussions/9913>

# ADR 047: Extend Upgrade Plan

## Changelog

- Nov, 23, 2021: Initial Draft
- May, 16, 2023: Proposal ABANDONED. `pre_run` and `post_run` are not necessary anymore and adding the `artifacts` brings minor benefits.

## Status

ABANDONED

## Abstract

This ADR expands the existing `x/upgrade Plan` proto message to include new fields for defining pre-run and post-run processes within upgrade tooling. It also defines a structure for providing downloadable artifacts involved in an upgrade.

## Context

The `upgrade` module in conjunction with Cosmovisor are designed to facilitate and automate a blockchain's transition from one version to another.

Users submit a software upgrade governance proposal containing an upgrade `Plan`. The [Plan](#) currently contains the following fields:

- `name` : A short string identifying the new version.
- `height` : The chain height at which the upgrade is to be performed.
- `info` : A string containing information about the upgrade.

The `info` string can be anything. However, Cosmovisor will try to use the `info` field to automatically download a new version of the blockchain executable. For the auto-download to work, Cosmovisor expects it to be either a stringified JSON object (with a specific structure defined through documentation), or a URL that will return such JSON. The JSON object identifies URLs used to download the new blockchain executable for different platforms (OS and Architecture, e.g. "linux/amd64"). Such a URL can either return the executable file directly or can return an archive containing the executable and possibly other assets.

If the URL returns an archive, it is decompressed into `{DAEMON_HOME}/cosmovisor/{upgrade name}`. Then, if `{DAEMON_HOME}/cosmovisor/{upgrade name}/bin/{DAEMON_NAME}` does not exist, but

`{DAEMON_HOME}/cosmovisor/{upgrade_name}/{DAEMON_NAME}` does, the latter is copied to the former.

If the URL returns something other than an archive, it is downloaded to

`{DAEMON_HOME}/cosmovisor/{upgrade_name}/bin/{DAEMON_NAME}`.

If an upgrade height is reached and the new version of the executable version isn't available, Cosmovisor will stop running.

Both `DAEMON_HOME` and `DAEMON_NAME` are [environment variables used to configure Cosmovisor](#).

Currently, there is no mechanism that makes Cosmovisor run a command after the upgraded chain has been restarted.

The current upgrade process has this timeline:

1. An upgrade governance proposal is submitted and approved.
2. The upgrade height is reached.
3. The `x/upgrade` module writes the `upgrade_info.json` file.
4. The chain halts.
5. Cosmovisor backs up the data directory (if set up to do so).
6. Cosmovisor downloads the new executable (if not already in place).
7. Cosmovisor executes the  `${DAEMON_NAME} pre-upgrade`.
8. Cosmovisor restarts the app using the new version and same args originally provided.

## Decision

### Protobuf Updates

We will update the `x/upgrade.Plan` message for providing upgrade instructions. The upgrade instructions will contain a list of artifacts available for each platform. It allows for the definition of a pre-run and post-run commands. These commands are not consensus guaranteed; they will be executed by Cosmovisor (or other) during its upgrade handling.

```
message Plan {
 // ... (existing fields)

 UpgradeInstructions instructions = 6;
}
```

The new `UpgradeInstructions instructions` field MUST be optional.

```
message UpgradeInstructions {
 string pre_run = 1;
 string post_run = 2;
 repeated Artifact artifacts = 3;
 string description = 4;
}
```

All fields in the `UpgradeInstructions` are optional.

- `pre_run` is a command to run prior to the upgraded chain restarting. If defined, it will be executed after halting and downloading the new artifact but before restarting the upgraded chain. The working directory this command runs from MUST be `{DAEMON_HOME}/cosmovisor/{upgrade}`

`name}`. This command MUST behave the same as the current [pre-upgrade](#) command. It does not take in any command-line arguments and is expected to terminate with the following exit codes:

| Exit status code                                                                                | How it is handled in Cosmosvisor                                                                                           |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| 0                                                                                               | Assumes <code>pre-upgrade</code> command executed successfully and continues the upgrade.                                  |
| 1                                                                                               | Default exit code when <code>pre-upgrade</code> command has not been implemented.                                          |
| 30                                                                                              | <code>pre-upgrade</code> command was executed but failed. This fails the entire upgrade.                                   |
| 31                                                                                              | <code>pre-upgrade</code> command was executed but failed. But the command is retried until exit code 1 or 30 are returned. |
| If defined, then the app supervisors (e.g. Cosmosvisor) MUST NOT run <code>app pre-run</code> . |                                                                                                                            |

- `post_run` is a command to run after the upgraded chain has been started. If defined, this command MUST be only executed at most once by an upgrading node. The output and exit code SHOULD be logged but SHOULD NOT affect the running of the upgraded chain. The working directory this command runs from MUST be `{DAEMON_HOME}/cosmosvisor/{upgrade name}`.
- `artifacts` define items to be downloaded. It SHOULD have only one entry per platform.
- `description` contains human-readable information about the upgrade and might contain references to external resources. It SHOULD NOT be used for structured processing information.

```
message Artifact {
 string platform = 1;
 string url = 2;
 string checksum = 3;
 string checksum_algo = 4;
}
```

- `platform` is a required string that SHOULD be in the format `{OS}/{CPU}`, e.g. "linux/amd64". The string "any" SHOULD also be allowed. An `Artifact` with a `platform` of "any" SHOULD be used as a fallback when a specific `{OS}/{CPU}` entry is not found. That is, if an `Artifact` exists with a `platform` that matches the system's OS and CPU, that should be used; otherwise, if an `Artifact` exists with a `platform` of `any`, that should be used; otherwise no artifact should be downloaded.
- `url` is a required URL string that MUST conform to [RFC 1738: Uniform Resource Locators](#). A request to this `url` MUST return either an executable file or an archive containing either `bin/{DAEMON_NAME}` or `{DAEMON_NAME}`. The URL should not contain checksum - it should be specified by the `checksum` attribute.

- `checksum` is a checksum of the expected result of a request to the `url`. It is not required, but is recommended. If provided, it MUST be a hex encoded checksum string. Tools utilizing these `UpgradeInstructions` MUST fail if a `checksum` is provided but is different from the checksum of the result returned by the `url`.
- `checksum_algo` is a string identify the algorithm used to generate the `checksum`. Recommended algorithms: `sha256`, `sha512`. Algorithms also supported (but not recommended): `sha1`, `md5`. If a `checksum` is provided, a `checksum_algo` MUST also be provided.

A `url` is not required to contain a `checksum` query parameter. If the `url` does contain a `checksum` query parameter, the `checksum` and `checksum_algo` fields MUST also be populated, and their values MUST match the value of the query parameter. For example, if the `url` is "`https://example.com?checksum=md5:d41d8cd98f00b204e9800998ecf8427e`", then the `checksum` field must be "`d41d8cd98f00b204e9800998ecf8427e`" and the `checksum_algo` field must be "`md5`".

## Upgrade Module Updates

If an upgrade `Plan` does not use the new `UpgradeInstructions` field, existing functionality will be maintained. The parsing of the `info` field as either a URL or `binaries` JSON will be deprecated. During validation, if the `info` field is used as such, a warning will be issued, but not an error.

We will update the creation of the `upgrade-info.json` file to include the `UpgradeInstructions`.

We will update the optional validation available via CLI to account for the new `Plan` structure. We will add the following validation:

1. If `UpgradeInstructions` are provided:
  1. There MUST be at least one entry in `artifacts`.
  2. All of the `artifacts` MUST have a unique `platform`.
  3. For each `Artifact`, if the `url` contains a `checksum` query parameter:
    1. The `checksum` query parameter value MUST be in the format of `{checksum_algo}:{checksum}`.
    2. The `{checksum}` from the query parameter MUST equal the `checksum` provided in the `Artifact`.
    3. The `{checksum_algo}` from the query parameter MUST equal the `checksum_algo` provided in the `Artifact`.
2. The following validation is currently done using the `info` field. We will apply similar validation to the `UpgradeInstructions`. For each `Artifact`:
  1. The `platform` MUST have the format `{OS}/{CPU}` or be "any".
  2. The `url` field MUST NOT be empty.
  3. The `url` field MUST be a proper URL.
  4. A `checksum` MUST be provided either in the `checksum` field or as a query parameter in the `url`.
  5. If the `checksum` field has a value and the `url` also has a `checksum` query parameter, the two values MUST be equal.
  6. The `url` MUST return either a file or an archive containing either `bin/{DAEMON_NAME}` or `{DAEMON_NAME}`.
  7. If a `checksum` is provided (in the field or as a query param), the checksum of the result of the `url` MUST equal the provided checksum.

Downloading of an `Artifact` will happen the same way that URLs from `info` are currently downloaded.

## Cosmovisor Updates

If the `upgrade-info.json` file does not contain any `UpgradeInstructions`, existing functionality will be maintained.

We will update Cosmovisor to look for and handle the new `UpgradeInstructions` in `upgrade-info.json`. If the `UpgradeInstructions` are provided, we will do the following:

1. The `info` field will be ignored.
2. The `artifacts` field will be used to identify the artifact to download based on the `platform` that Cosmovisor is running in.
3. If a `checksum` is provided (either in the field or as a query param in the `url`), and the downloaded artifact has a different checksum, the upgrade process will be interrupted and Cosmovisor will exit with an error.
4. If a `pre_run` command is defined, it will be executed at the same point in the process where the `app pre-upgrade` command would have been executed. It will be executed using the same environment as other commands run by Cosmovisor.
5. If a `post_run` command is defined, it will be executed after executing the command that restarts the chain. It will be executed in a background process using the same environment as the other commands. Any output generated by the command will be logged. Once complete, the exit code will be logged.

We will deprecate the use of the `info` field for anything other than human readable information. A warning will be logged if the `info` field is used to define the assets (either by URL or JSON).

The new upgrade timeline is very similar to the current one. Changes are in bold:

1. An upgrade governance proposal is submitted and approved.
2. The upgrade height is reached.
3. The `x/upgrade` module writes the `upgrade_info.json` file (**now possibly with `UpgradeInstructions`**).
4. The chain halts.
5. Cosmovisor backs up the data directory (if set up to do so).
6. Cosmovisor downloads the new executable (if not already in place).
7. Cosmovisor executes the **`pre_run` command if provided**, or else the `$(DAEMON_NAME) pre-upgrade` command.
8. Cosmovisor restarts the app using the new version and same args originally provided.
9. **Cosmovisor immediately runs the `post_run` command in a detached process.**

## Consequences

### Backwards Compatibility

Since the only change to existing definitions is the addition of the `instructions` field to the `Plan` message, and that field is optional, there are no backwards incompatibilities with respects to the proto messages. Additionally, current behavior will be maintained when no `UpgradeInstructions` are provided, so there are no backwards incompatibilities with respects to either the upgrade module or Cosmovisor.

### Forwards Compatibility

In order to utilize the `UpgradeInstructions` as part of a software upgrade, both of the following must be true:

1. The chain must already be using a sufficiently advanced version of the Cosmos SDK.
2. The chain's nodes must be using a sufficiently advanced version of Cosmovisor.

## Positive

1. The structure for defining artifacts is clearer since it is now defined in the proto instead of in documentation.
2. Availability of a pre-run command becomes more obvious.
3. A post-run command becomes possible.

## Negative

1. The `Plan` message becomes larger. This is negligible because A) the `x/upgrades` module only stores at most one upgrade plan, and B) upgrades are rare enough that the increased gas cost isn't a concern.
2. There is no option for providing a URL that will return the `UpgradeInstructions`.
3. The only way to provide multiple assets (executables and other files) for a platform is to use an archive as the platform's artifact.

## Neutral

1. Existing functionality of the `info` field is maintained when the `UpgradeInstructions` aren't provided.

## Further Discussions

1. [Draft PR #10032 Comment](#): Consider different names for `UpgradeInstructions` instructions (either the message type or field name).
2. [Draft PR #10032 Comment](#):
  1. Consider putting the `string platform` field inside `UpgradeInstructions` and make `UpgradeInstructions` a repeated field in `Plan`.
  2. Consider using a `oneof` field in the `Plan` which could either be `UpgradeInstructions` or else a URL that should return the `UpgradeInstructions`.
  3. Consider allowing `info` to either be a JSON serialized version of `UpgradeInstructions` or else a URL that returns that.
3. [Draft PR #10032 Comment](#): Consider not including the `UpgradeInstructions.description` field, using the `info` field for that purpose instead.
4. [Draft PR #10032 Comment](#): Consider allowing multiple artifacts to be downloaded for any given platform by adding a `name` field to the `Artifact` message.
5. [PR #10502 Comment](#) Allow the new `UpgradeInstructions` to be provided via URL.
6. [PR #10502 Comment](#) Allow definition of a `signer` for assets (as an alternative to using a `checksum`).

## References

- [Current upgrade.proto](#)
- [Upgrade Module README](#)
- [Cosmovisor README](#)
- [Pre-upgrade README](#)
- [Draft/POC PR #10032](#)

- [RFC 1738: Uniform Resource Locators](#)

# ADR 048: Multi Tire Gas Price System

## Changelog

- Dec 1, 2021: Initial Draft

## Status

Rejected

## Abstract

This ADR describes a flexible mechanism to maintain a consensus level gas prices, in which one can choose a multi-tier gas price system or EIP-1559 like one through configuration.

## Context

Currently, each validator configures it's own `minimal-gas-prices` in `app.yaml`. But setting a proper minimal gas price is critical to protect network from dos attack, and it's hard for all the validators to pick a sensible value, so we propose to maintain a gas price in consensus level.

Since tendermint 0.34.20 has supported mempool prioritization, we can take advantage of that to implement more sophisticated gas fee system.

## Multi-Tier Price System

We propose a multi-tier price system on consensus to provide maximum flexibility:

- Tier 1: a constant gas price, which could only be modified occasionally through governance proposal.
- Tier 2: a dynamic gas price which is adjusted according to previous block load.
- Tier 3: a dynamic gas price which is adjusted according to previous block load at a higher speed.

The gas price of higher tier should bigger than the lower tier.

The transaction fees are charged with the exact gas price calculated on consensus.

The parameter schema is like this:

```
message TierParams {
 uint32 priority = 1 // priority in tendermint mempool
 Coin initial_gas_price = 2 //
 uint32 parent_gas_target = 3 // the target saturation of block
 uint32 change_denominator = 4 // decides the change speed
 Coin min_gas_price = 5 // optional lower bound of the price adjustment
 Coin max_gas_price = 6 // optional upper bound of the price adjustment
}

message Params {
```

```
 repeated TierParams tiers = 1;
}
```

## Extension Options

We need to allow user to specify the tier of service for the transaction, to support it in an extensible way, we add an extension option in `AuthInfo` :

```
message ExtensionOptionsTieredTx {
 uint32 fee_tier = 1
}
```

The value of `fee_tier` is just the index to the `tiers` parameter list.

We also change the semantic of existing `fee` field of `Tx`, instead of charging user the exact `fee` amount, we treat it as a fee cap, while the actual amount of fee charged is decided dynamically. If the `fee` is smaller than dynamic one, the transaction won't be included in current block and ideally should stay in the mempool until the consensus gas price drop. The mempool can eventually prune old transactions.

## Tx Prioritization

Transactions are prioritized based on the tier, the higher the tier, the higher the priority.

Within the same tier, follow the default Tendermint order (currently FIFO). Be aware of that the mempool tx ordering logic is not part of consensus and can be modified by malicious validator.

This mechanism can be easily composed with prioritization mechanisms:

- we can add extra tiers out of a user control:
  - Example 1: user can set tier 0, 10 or 20, but the protocol will create tiers 0, 1, 2 ... 29. For example IBC transactions will go to tier `user_tier + 5` : if user selected tier 1, then the transaction will go to tier 15.
  - Example 2: we can reserve tier 4, 5, ... only for special transaction types. For example, tier 5 is reserved for evidence tx. So if submits a bank.Send transaction and set tier 5, it will be delegated to tier 3 (the max tier level available for any transaction).
  - Example 3: we can enforce that all transactions of a specific type will go to specific tier. For example, tier 100 will be reserved for evidence transactions and all evidence transactions will always go to that tier.

### min-gas-prices

Deprecate the current per-validator `min-gas-prices` configuration, since it would confusing for it to work together with the consensus gas price.

## Adjust For Block Load

For tier 2 and tier 3 transactions, the gas price is adjusted according to previous block load, the logic could be similar to EIP-1559:

```
def adjust_gas_price(gas_price, parent_gas_used, tier):
 if parent_gas_used == tier.parent_gas_target:
 return gas_price
```

```

 elif parent_gas_used > tier.parent_gas_target:
 gas_used_delta = parent_gas_used - tier.parent_gas_target
 gas_price_delta = max(gas_price * gas_used_delta // tier.parent_gas_target // tier.change_speed, 1)
 return gas_price + gas_price_delta
 else:
 gas_used_delta = parent_gas_target - parent_gas_used
 gas_price_delta = gas_price * gas_used_delta // parent_gas_target // tier.change_speed
 return gas_price - gas_price_delta

```

## Block Segment Reservation

Ideally we should reserve block segments for each tier, so the lower tiered transactions won't be completely squeezed out by higher tier transactions, which will force user to use higher tier, and the system degraded to a single tier.

We need help from tendermint to implement this.

## Implementation

We can make each tier's gas price strategy fully configurable in protocol parameters, while providing a sensible default one.

Pseudocode in python-like syntax:

```

interface TieredTx:
 def tier(self) -> int:
 pass

 def tx_tier(tx):
 if isinstance(tx, TieredTx):
 return tx.tier()
 else:
 # default tier for custom transactions
 return 0
 # NOTE: we can add more rules here per "Tx Prioritization" section

class TierParams:
 'gas price strategy parameters of one tier'
 priority: int # priority in tendermint mempool
 initial_gas_price: Coin
 parent_gas_target: int
 change_speed: Decimal # 0 means don't adjust for block load.

class Params:
 'protocol parameters'
 tiers: List[TierParams]

class State:
 'consensus state'
 # total gas used in last block, None when it's the first block

```

```

parent_gas_used: Optional[int]
 # gas prices of last block for all tiers
gas_prices: List[Coin]

def begin_block():
 'Adjust gas prices'
 for i, tier in enumerate(Params.tiers):
 if State.parent_gas_used is None:
 # initialized gas price for the first block
 State.gas_prices[i] = tier.initial_gas_price
 else:
 # adjust gas price according to gas used in previous block
 State.gas_prices[i] = adjust_gas_price(State.gas_prices[i],
State.parent_gas_used, tier)

def mempoolFeeTxHandler_checkTx(ctx, tx):
 # the minimal-gas-price configured by validator, zero in deliver_tx context
 validator_price = ctx.MinGasPrice()
 consensus_price = State.gas_prices[tx_tier(tx)]
 min_price = max(validator_price, consensus_price)

 # zero means infinity for gas price cap
 if tx.gas_price() > 0 and tx.gas_price() < min_price:
 return 'insufficient fees'
 return next_CheckTx(ctx, tx)

def txPriorityHandler_checkTx(ctx, tx):
 res, err := next_CheckTx(ctx, tx)
 # pass priority to tendermint
 res.Priority = Params.tiers[tx_tier(tx)].priority
 return res, err

def end_block():
 'Update block gas used'
 State.parent_gas_used = block_gas_meter.consumed()

```

## Dos attack protection

To fully saturate the blocks and prevent other transactions from executing, attacker need to use transactions of highest tier, the cost would be significantly higher than the default tier.

If attacker spam with lower tier transactions, user can mitigate by sending higher tier transactions.

## Consequences

### Backwards Compatibility

- New protocol parameters.
- New consensus states.
- New/changed fields in transaction body.

### Positive

- The default tier keeps the same predictable gas price experience for client.
- The higher tier's gas price can adapt to block load.
- No priority conflict with custom priority based on transaction types, since this proposal only occupy three priority levels.
- Possibility to compose different priority rules with tiers

## Negative

- Wallets & tools need to update to support the new `tier` parameter, and semantic of `fee` field is changed.

## Neutral

## References

- <https://eips.ethereum.org/EIPS/eip-1559>
- <https://iohk.io/en/blog/posts/2021/11/26/network-traffic-and-tiered-pricing/>

# ADR 049: State Sync Hooks

## Changelog

- Jan 19, 2022: Initial Draft
- Apr 29, 2022: Safer extension snapshotter interface

## Status

Implemented

## Abstract

This ADR outlines a hooks-based mechanism for application modules to provide additional state (outside of the IAVL tree) to be used during state sync.

## Context

New clients use state-sync to download snapshots of module state from peers. Currently, the snapshot consists of a stream of `SnapshotStoreItem` and `SnapshotIAVLItem`, which means that application modules that define their state outside of the IAVL tree cannot include their state as part of the state-sync process.

Note, Even though the module state data is outside of the tree, for determinism we require that the hash of the external data should be posted in the IAVL tree.

## Decision

A simple proposal based on our existing implementation is that, we can add two new message types: `SnapshotExtensionMeta` and `SnapshotExtensionPayload`, and they are appended to the existing multi-store stream with `SnapshotExtensionMeta` acting as a delimiter between extensions. As the chunk hashes should be able to ensure data integrity, we don't need a delimiter to mark the end of the snapshot stream.

Besides, we provide `Snapshotter` and `ExtensionSnapshotter` interface for modules to implement snapshotters, which will handle both taking snapshot and the restoration. Each module could have multiple snapshotters, and for modules with additional state, they should implement `ExtensionSnapshotter` as extension snapshotters. When setting up the application, the `snapshot Manager` should call `RegisterExtensions([]ExtensionSnapshotter...)` to register all the extension snapshotters.

```
// SnapshotItem is an item contained in a rootmulti.Store snapshot.
// On top of the existing SnapshotStoreItem and SnapshotIAVListItem, we add two new
options for the item.

message SnapshotItem {
 // item is the specific type of snapshot item.
 oneof item {
 SnapshotStoreItem store = 1;
 SnapshotIAVListItem iavl = 2 [(gogoproto.customname) =
"IAVL"];
 SnapshotExtensionMeta extension = 3;
 SnapshotExtensionPayload extension_payload = 4;
 }
}

// SnapshotExtensionMeta contains metadata about an external snapshotter.
// One module may need multiple snapshotters, so each module may have multiple
SnapshotExtensionMeta.

message SnapshotExtensionMeta {
 // the name of the ExtensionSnapshotter, and it is registered to snapshotter
manager when setting up the application
 // name should be unique for each ExtensionSnapshotter as we need to
alphabetically order their snapshots to get
 // deterministic snapshot stream.
 string name = 1;
 // this is used by each ExtensionSnapshotter to decide the format of payloads
included in SnapshotExtensionPayload message
 // it is used within the snapshotter/namespace, not global one for all modules
 uint32 format = 2;
}

// SnapshotExtensionPayload contains payloads of an external snapshotter.

message SnapshotExtensionPayload {
 bytes payload = 1;
}
```

When we create a snapshot stream, the `multistore` snapshot is always placed at the beginning of the binary stream, and other extension snapshots are alphabetically ordered by the name of the corresponding `ExtensionSnapshotter`.

The snapshot stream would look like as follows:

```
// multi-store snapshot
{SnapshotStoreItem | SnapshotIAVListItem, ...}
// extension1 snapshot
```

```

SnapshotExtensionMeta
{SnapshotExtensionPayload, ...}
// extension2 snapshot
SnapshotExtensionMeta
{SnapshotExtensionPayload, ...}

```

We add an `extensions` field to `snapshot Manager` for extension snapshotters. The `multistore` snapshotter is a special one and it doesn't need a name because it is always placed at the beginning of the binary stream.

```

type Manager struct {
 store *Store
 multistore types.Snapshotter
 extensions map[string]types.ExtensionSnapshotter
 mtx sync.Mutex
 operation operation
 chRestore chan<- io.ReadCloser
 chRestoreDone <-chan restoreDone
 restoreChunkHashes [][]byte
 restoreChunkIndex uint32
}

```

For extension snapshotters that implement the `ExtensionSnapshotter` interface, their names should be registered to the `snapshot Manager` by calling `RegisterExtensions` when setting up the application. The snapshotters will handle both taking snapshot and restoration.

```

// RegisterExtensions register extension snapshotters to manager
func (m *Manager) RegisterExtensions(extensions ...types.ExtensionSnapshotter) error

```

On top of the existing `Snapshotter` interface for the `multistore`, we add `ExtensionSnapshotter` interface for the extension snapshotters. Three more function signatures: `SnapshotFormat()`, `SupportedFormats()` and `SnapshotName()` are added to `ExtensionSnapshotter`.

```

// ExtensionPayloadReader read extension payloads,
// it returns io.EOF when reached either end of stream or the extension boundaries.
type ExtensionPayloadReader = func() ([]byte, error)

// ExtensionPayloadWriter is a helper to write extension payloads to underlying
// stream.
type ExtensionPayloadWriter = func([]byte) error

// ExtensionSnapshotter is an extension Snapshotter that is appended to the snapshot
// stream.
// ExtensionSnapshotter has an unique name and manages it's own internal formats.
type ExtensionSnapshotter interface {
 // SnapshotName returns the name of snapshotter, it should be unique in the
 // manager.
 SnapshotName() string
}

```

```

// SnapshotFormat returns the default format used to take a snapshot.
SnapshotFormat() uint32

// SupportedFormats returns a list of formats it can restore from.
SupportedFormats() []uint32

// SnapshotExtension writes extension payloads into the underlying protobuf
stream.
SnapshotExtension(height uint64, payloadWriter ExtensionPayloadWriter) error

// RestoreExtension restores an extension state snapshot,
// the payload reader returns `io.EOF` when reached the extension boundaries.
RestoreExtension(height uint64, format uint32, payloadReader
ExtensionPayloadReader) error

}

```

## Consequences

As a result of this implementation, we are able to create snapshots of binary chunk stream for the state that we maintain outside of the IAVL Tree, CosmWasm blobs for example. And new clients are able to fetch snapshots of state for all modules that have implemented the corresponding interface from peer nodes.

### Backwards Compatibility

This ADR introduces new proto message types, add an `extensions` field in `snapshot Manager`, and add new `ExtensionSnapshotter` interface, so this is not backwards compatible if we have extensions.

But for applications that does not have the state data outside of the IAVL tree for any module, the snapshot stream is backwards-compatible.

### Positive

- State maintained outside of IAVL tree like CosmWasm blobs can create snapshots by implementing extension snapshotters, and being fetched by new clients via state-sync.

### Negative

### Neutral

- All modules that maintain state outside of IAVL tree need to implement `ExtensionSnapshotter` and the `snapshot Manager` need to call `RegisterExtensions` when setting up the application.

## Further Discussions

While an ADR is in the DRAFT or PROPOSED stage, this section should contain a summary of issues to be solved in future iterations (usually referencing comments from a pull-request discussion). Later, this section can optionally list ideas or improvements the author or reviewers found during the analysis of this ADR.

## Test Cases [optional]

Test cases for an implementation are mandatory for ADRs that are affecting consensus changes. Other ADRs can choose to include links to test cases if applicable.

## References

- <https://github.com/cosmos/cosmos-sdk/pull/10961>
- <https://github.com/cosmos/cosmos-sdk/issues/7340>
- <https://hackmd.io/gJoyev6DSmqgkO667WQIGw>

# ADR 050: SIGN\_MODE\_TEXTUAL: Annex 1 Value Renderers

## Changelog

- Dec 06, 2021: Initial Draft
- Feb 07, 2022: Draft read and concept-ACKed by the Ledger team.
- Dec 01, 2022: Remove `Object:` prefix on Any header screen.
- Dec 13, 2022: Sign over bytes hash when bytes length > 32.
- Mar 27, 2023: Update `Any` value renderer to omit message header screen.

## Status

Accepted. Implementation started. Small value renderers details still need to be polished.

## Abstract

This Annex describes value renderers, which are used for displaying Protobuf values in a human-friendly way using a string array.

## Value Renderers

Value Renderers describe how values of different Protobuf types should be encoded as a string array. Value renderers can be formalized as a set of bijective functions `func renderT(value T) []string`, where `T` is one of the below Protobuf types for which this spec is defined.

### Protobuf number

- Applies to:
  - protobuf numeric integer types (`int{32,64}`, `uint{32,64}`, `sint{32,64}`,  
`fixed{32,64}`, `sfixed{32,64}`)
  - strings whose `customtype` is `github.com/cosmos/cosmos-sdk/types.Int` or  
`github.com/cosmos/cosmos-sdk/types.Dec`
  - bytes whose `customtype` is `github.com/cosmos/cosmos-sdk/types.Int` or  
`github.com/cosmos/cosmos-sdk/types.Dec`
- Trailing decimal zeroes are always removed
- Formatting with `,` s for every three integral digits.
- Usage of `.` to denote the decimal delimiter.

### Examples

- `1000 (uint64)` -> `1'000`
- `"1000000.00"` (string representing a Dec) -> `1'000'000`

- "1000000.10" (string representing a Dec) -> 1'000'000.1

### coin

- Applies to `cosmos.base.v1beta1.Coin`.
- Denoms are converted to `display` denoms using `Metadata` (if available). **This requires a state query.** The definition of `Metadata` can be found in the [bank protobuf definition](#). If the `display` field is empty or nil, then we do not perform any denom conversion.
- Amounts are converted to `display` denom amounts and rendered as `number`s above
  - We do not change the capitalization of the denom. In practice, `display` denoms are stored in lowercase in state (e.g. `10 atom`), however they are often showed in UPPERCASE in everyday life (e.g. `10 ATOM`). Value renderers keep the case used in state, but we may recommend chains changing the denom metadata to be uppercase for better user display.
- One space between the denom and amount (e.g. `10 atom`).
- In the future, IBC denoms could maybe be converted to DID/IIDs, if we can find a robust way for doing this (ex. `cosmos:cosmos:hub:bank:denom:atom`)

### Examples

- `1000000000uatom` -> `["1'000 atom"]`, because atom is the metadata's display denom.

### coins

- an array of `coin` is display as the concatenation of each `coin` encoded as the specification above, the joined together with the delimiter `", "` (a comma and a space, no quotes around).
- the list of coins is ordered by unicode code point of the display denom: `A-Z < a-z`. For example, the string `aAbBcC` would be sorted `ABCabc`.
  - if the coins list had 0 items in it then it'll be rendered as `zero`

### Example

- `["3cosm", "2000000uatom"]` -> `2 atom, 3 COSM` (assuming the display denoms are `atom` and `COSM`)
- `["10atom", "20Acoin"]` -> `20 Acoin, 10 atom` (assuming the display denoms are `atom` and `Acoin`)
- `[]` -> `zero`

### repeated

- Applies to all `repeated` fields, except `cosmos.tx.v1beta1.TxBODY#Messages`, which has a particular encoding (see [ADR-050](#)).
- A repeated type has the following template:

```
<field_name>: <int> <field_kind>
<field_name> (<index>/<int>): <value rendered 1st line>
<optional value rendered in the next lines>
<field_name> (<index>/<int>): <value rendered 1st line>
<optional value rendered in the next lines>
End of <field_name>.
```

where:

- `field_name` is the Protobuf field name of the repeated field
- `field_kind`:
  - if the type of the repeated field is a message, `field_kind` is the message name
  - if the type of the repeated field is an enum, `field_kind` is the enum name
  - in any other case, `field_kind` is the protobuf primitive type (e.g. "string" or "bytes")
- `int` is the length of the array
- `index` is one based index of the repeated field

## Examples

Given the proto definition:

```
message AllowedMsgAllowance {
 repeated string allowed_messages = 1;
}
```

and initializing with:

```
x := []AllowedMsgAllowance{"cosmos.bank.v1beta1.MsgSend", "cosmos.gov.v1.MsgVote"}
```

we have the following value-rendered encoding:

```
Allowed messages: 2 strings
Allowed messages (1/2): cosmos.bank.v1beta1.MsgSend
Allowed messages (2/2): cosmos.gov.v1.MsgVote
End of Allowed messages
```

## message

- Applies to all Protobuf messages that do not have a custom encoding.
- Field names follow [sentence case](#)
  - replace each `_` with a space
  - capitalize first letter of the sentence
- Field names are ordered by their Protobuf field number
- Screen title is the field name, and screen content is the value.
- Nesting:
  - if a field contains a nested message, we value-render the underlying message using the template:

```
<field_name>: <1st line of value-rendered message>
> <lines 2-n of value-rendered message> // Notice the `>` prefix.

 ◦ > character is used to denote nesting. For each additional level of nesting, add >.
```

## Examples

Given the following Protobuf messages:

```

enum VoteOption {
 VOTE_OPTION_UNSPECIFIED = 0;
 VOTE_OPTION_YES = 1;
 VOTE_OPTION_ABSTAIN = 2;
 VOTE_OPTION_NO = 3;
 VOTE_OPTION_NO_WITH_VETO = 4;
}

message WeightedVoteOption {
 VoteOption option = 1;
 string weight = 2 [(cosmos_proto.scalar) = "cosmos.Dec"];
}

message Vote {
 uint64 proposal_id = 1;
 string voter = 2 [(cosmos_proto.scalar) = "cosmos.AddressString"];
 reserved 3;
 repeated WeightedVoteOption options = 4;
}

```

we get the following encoding for the `Vote` message:

```

Vote object
> Proposal id: 4
> Voter: cosmoslabc...def
> Options: 2 WeightedVoteOptions
> Options (1/2): WeightedVoteOption object
>> Option: VOTE_OPTION_YES
>> Weight: 0.7
> Options (2/2): WeightedVoteOption object
>> Option: VOTE_OPTION_NO
>> Weight: 0.3
> End of Options

```

## Enums

- Show the enum variant name as string.

## Examples

See example above with `message Vote{}`.

### `google.protobuf.Any`

- Applies to `google.protobuf.Any`
- Rendered as:

```

<type_url>
> <value rendered underlying message>

```

There is however one exception: when the underlying message is a Protobuf message that does not have a custom encoding, then the message header screen is omitted, and one level of indentation is removed.

Messages that have a custom encoding, including `google.protobuf.Timestamp`, `google.protobuf.Duration`, `google.protobuf.Any`, `cosmos.base.v1beta1.Coin`, and messages that have an app-defined custom encoding, will preserve their header and indentation level.

## Examples

Message header screen is stripped, one-level of indentation removed:

```
/cosmos.gov.v1.Vote
> Proposal id: 4
> Vote: cosmoslabc...def
> Options: 2 WeightedVoteOptions
> Options (1/2): WeightedVoteOption object
>> Option: Yes
>> Weight: 0.7
> Options (2/2): WeightedVoteOption object
>> Option: No
>> Weight: 0.3
> End of Options
```

Message with custom encoding:

```
/cosmos.base.v1beta1.Coin
> 10uatom
```

### `google.protobuf.Timestamp`

Rendered using [RFC 3339](#) (a simplification of ISO 8601), which is the current recommendation for portable time values. The rendering always uses "Z" (UTC) as the timezone. It uses only the necessary fractional digits of a second, omitting the fractional part entirely if the timestamp has no fractional seconds. (The resulting timestamps are not automatically sortable by standard lexicographic order, but we favor the legibility of the shorter string.)

## Examples

The timestamp with 1136214245 seconds and 700000000 nanoseconds is rendered as `2006-01-02T15:04:05.7Z`. The timestamp with 1136214245 seconds and zero nanoseconds is rendered as `2006-01-02T15:04:05Z`.

### `google.protobuf.Duration`

The duration proto expresses a raw number of seconds and nanoseconds. This will be rendered as longer time units of days, hours, and minutes, plus any remaining seconds, in that order. Leading and trailing zero-quantity units will be omitted, but all units in between nonzero units will be shown, e.g. `3 days, 0 hours, 0 minutes, 5 seconds`.

Even longer time units such as months or years are imprecise. Weeks are precise, but not commonly used - `91 days` is more immediately legible than `13 weeks`. Although `days` can be problematic, e.g. noon to noon on subsequent days can be 23 or 25 hours depending on daylight savings transitions, there is significant advantage in using strict 24-hour days over using only hours (e.g. `91 days` vs `2184 hours`).

When nanoseconds are nonzero, they will be shown as fractional seconds, with only the minimum number of digits, e.g. `0.5 seconds`.

A duration of exactly zero is shown as `0 seconds`.

Units will be given as singular (no trailing `s`) when the quantity is exactly one, and will be shown in plural otherwise.

Negative durations will be indicated with a leading minus sign (`-`).

Examples:

- `1 day`
- `30 days`
- `-1 day, 12 hours`
- `3 hours, 0 minutes, 53.025 seconds`

## bytes

- Bytes of length shorter or equal to 35 are rendered in hexadecimal, all capital letters, without the `0x` prefix.
- Bytes of length greater than 35 are hashed using SHA256. The rendered text is `SHA-256=`, followed by the 32-byte hash, in hexadecimal, all capital letters, without the `0x` prefix.
- The hexadecimal string is finally separated into groups of 4 digits, with a space `' '` as separator. If the bytes length is odd, the 2 remaining hexadecimal characters are at the end.

The number 35 was chosen because it is the longest length where the hashed-and-prefixed representation is longer than the original data directly formatted, using the 3 rules above. More specifically:

- a 35-byte array will have 70 hex characters, plus 17 space characters, resulting in 87 characters.
- byte arrays starting from length 36 will be hashed to 32 bytes, which is 64 hex characters plus 15 spaces, and with the `SHA-256=` prefix, it takes 87 characters. Also, secp256k1 public keys have length 33, so their Textual representation is not their hashed value, which we would like to avoid.

Note: Data longer than 35 bytes are not rendered in a way that can be inverted. See ADR-050's [section about invertability](#) for a discussion.

Examples

Inputs are displayed as byte arrays.

- `[0] : 00`
- `[0,1,2] : 0001 02`
- `[0,1,2,...,34] : 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F 1011 1213 1415 1617 1819 1A1B 1C1D 1E1F 2021 22`
- `[0,1,2,...,35] : SHA-256=5D7E 2D9B 1DCB C85E 7C89 0036 A2CF 2F9F E7B6 6554 F2DF 08CE C6AA 9C0A 25C9 9C21`

## address bytes

We currently use `string` types in protobuf for addresses so this may not be needed, but if any address bytes are used in sign mode textual they should be rendered with bech32 formatting

## strings

Strings are rendered as-is.

## Default Values

- Default Protobuf values for each field are skipped.

### Example

```
message TestData {
 string signer = 1;
 string metadata = 2;
}
```

```
myTestData := TestData{
 Signer: "cosmos1abc"
}
```

We get the following encoding for the `TestData` message:

```
TestData object
> Signer: cosmos1abc
```

### bool

Boolean values are rendered as `True` or `False`.

### [ABANDONED] Custom `msg_title` instead of `Msg type_url`

*This paragraph is in the Annex for informational purposes only, and will be removed in a next update of the ADR.*

- ▶ Click to see abandoned idea.

# ADR 050: SIGN\_MODE\_TEXTUAL: Annex 2 XXX

## Changelog

- Oct 3, 2022: Initial Draft

## Status

DRAFT

## Abstract

This annex provides normative guidance on how devices should render a `SIGN_MODE_TEXTUAL` document.

## Context

`SIGN_MODE_TEXTUAL` allows a legible version of a transaction to be signed on a hardware security device, such as a Ledger. Early versions of the design rendered transactions directly to lines of ASCII text, but this proved awkward from its in-band signaling, and for the need to display Unicode text within the transaction.

## Decision

`SIGN_MODE_TEXTUAL` renders to an abstract representation, leaving it up to device-specific software how to present this representation given the capabilities, limitations, and conventions of the device.

We offer the following normative guidance:

1. The presentation should be as legible as possible to the user, given the capabilities of the device. If legibility could be sacrificed for other properties, we would recommend just using some other signing mode. Legibility should focus on the common case - it is okay for unusual cases to be less legible.
2. The presentation should be invertible if possible without substantial sacrifice of legibility. Any change to the rendered data should result in a visible change to the presentation. This extends the integrity of the signing to user-visible presentation.
3. The presentation should follow normal conventions of the device, without sacrificing legibility or invertibility.

As an illustration of these principles, here is an example algorithm for presentation on a device which can display a single 80-character line of printable ASCII characters:

- The presentation is broken into lines, and each line is presented in sequence, with user controls for going forward or backward a line.
- Expert mode screens are only presented if the device is in expert mode.
- Each line of the screen starts with a number of `>` characters equal to the screen's indentation level, followed by a `+` character if this isn't the first line of the screen, followed by a space if either a `>` or a `+` has been emitted, or if this header is followed by a `,`, `+`, or space.
- If the line ends with whitespace or an `@` character, an additional `@` character is appended to the line.
- The following ASCII control characters or backslash (`\`) are converted to a backslash followed by a letter code, in the manner of string literals in many languages:
  - a: U+0007 alert or bell
  - b: U+0008 backspace
  - f: U+000C form feed
  - n: U+000A line feed
  - r: U+000D carriage return
  - t: U+0009 horizontal tab
  - v: U+000B vertical tab
  - `\`: U+005C backslash
- All other ASCII control characters, plus non-ASCII Unicode code points, are shown as either:
  - `\u` followed by 4 uppercase hex characters for code points in the basic multilingual plane (BMP).
  - `\U` followed by 8 uppercase hex characters for other code points.
- The screen will be broken into multiple lines to fit the 80-character limit, considering the above transformations in a way that attempts to minimize the number of lines generated. Expanded control or Unicode characters are never split across lines.

Example output:

```

An introductory line.

key1: 123456
key2: a string that ends in whitespace @
key3: a string that ends in a single ampersand - @@
>tricky key4<: note the leading space in the presentation
introducing an aggregate
> key5: false
> key6: a very long line of text, please co\u00F6perate and break into
>+ multiple lines.
> Can we do further nesting?
>> You bet we can!

```

The inverse mapping gives us the only input which could have generated this output (JSON notation for string data):

```

Indent Text
----- -----
0 "An introductory line."
0 "key1: 123456"
0 "key2: a string that ends in whitespace "
0 "key3: a string that ends in a single ampersand - @"
0 ">tricky key4<: note the leading space in the presentation"
0 "introducing an aggregate"
1 "key5: false"
1 "key6: a very long line of text, please coöperate and break into multiple
lines."
1 "Can we do further nesting?"
2 "You bet we can!"

```

## ADR 050: SIGN\_MODE\_TEXTUAL

### Changelog

- Dec 06, 2021: Initial Draft.
- Feb 07, 2022: Draft read and concept-ACKed by the Ledger team.
- May 16, 2022: Change status to Accepted.
- Aug 11, 2022: Require signing over tx raw bytes.
- Sep 07, 2022: Add custom `Msg -renderers`.
- Sep 18, 2022: Structured format instead of lines of text
- Nov 23, 2022: Specify CBOR encoding.
- Dec 01, 2022: Link to examples in separate JSON file.
- Dec 06, 2022: Re-ordering of envelope screens.
- Dec 14, 2022: Mention exceptions for invertability.
- Jan 23, 2023: Switch Screen.Text to Title+Content.
- Mar 07, 2023: Change SignDoc from array to struct containing array.
- Mar 20, 2023: Introduce a spec version initialized to 0.

### Status

Accepted. Implementation started. Small value renderers details still need to be polished.

Spec version: 0.

## Abstract

This ADR specifies SIGN\_MODE\_TEXTUAL, a new string-based sign mode that is targetted at signing with hardware devices.

## Context

Protobuf-based SIGN\_MODE\_DIRECT was introduced in [ADR-020](#) and is intended to replace SIGN\_MODE\_LEGACY\_AMINO\_JSON in most situations, such as mobile wallets and CLI keyrings. However, the [Ledger](#) hardware wallet is still using SIGN\_MODE\_LEGACY\_AMINO\_JSON for displaying the sign bytes to the user. Hardware wallets cannot transition to SIGN\_MODE\_DIRECT as:

- SIGN\_MODE\_DIRECT is binary-based and thus not suitable for display to end-users. Technically, hardware wallets could simply display the sign bytes to the user. But this would be considered as blind signing, and is a security concern.
- hardware cannot decode the protobuf sign bytes due to memory constraints, as the Protobuf definitions would need to be embedded on the hardware device.

In an effort to remove Amino from the SDK, a new sign mode needs to be created for hardware devices.

[Initial discussions](#) propose a text-based sign mode, which this ADR formally specifies.

## Decision

In SIGN\_MODE\_TEXTUAL, a transaction is rendered into a textual representation, which is then sent to a secure device or subsystem for the user to review and sign. Unlike `SIGN_MODE_DIRECT`, the transmitted data can be simply decoded into legible text even on devices with limited processing and display.

The textual representation is a sequence of *screens*. Each screen is meant to be displayed in its entirety (if possible) even on a small device like a Ledger. A screen is roughly equivalent to a short line of text. Large screens can be displayed in several pieces, much as long lines of text are wrapped, so no hard guidance is given, though 40 characters is a good target. A screen is used to display a single key/value pair for scalar values (or composite values with a compact notation, such as `Coins`) or to introduce or conclude a larger grouping.

The text can contain the full range of Unicode code points, including control characters and nul. The device is responsible for deciding how to display characters it cannot render natively. See [annex 2](#) for guidance.

Screens have a non-negative indentation level to signal composite or nested structures. Indentation level zero is the top level. Indentation is displayed via some device-specific mechanism. Message quotation notation is an appropriate model, such as leading `>` characters or vertical bars on more capable displays.

Some screens are marked as *expert* screens, meant to be displayed only if the viewer chooses to opt in for the extra detail. Expert screens are meant for information that is rarely useful, or needs to be present only for signature integrity (see below).

## Invertible Rendering

We require that the rendering of the transaction be invertible: there must be a parsing function such that for every transaction, when rendered to the textual representation, parsing that representation yeilds a proto message equivalent to the original under proto equality.

Note that this inverse function does not need to perform correct parsing or error signaling for the whole domain of textual data. Merely that the range of valid transactions be invertible under the composition of rendering and parsing.

Note that the existence of an inverse function ensures that the rendered text contains the full information of the original transaction, not a hash or subset.

We make an exception for invertibility for data which are too large to meaningfully display, such as byte strings longer than 32 bytes. We may then selectively render them with a cryptographically-strong hash. In these cases, it is still computationally infeasible to find a different transaction which has the same rendering. However, we must ensure that the hash computation is simple enough to be reliably executed independently, so at least the hash is itself reasonably verifiable when the raw byte string is not.

## Chain State

The rendering function (and parsing function) may depend on the current chain state. This is useful for reading parameters, such as coin display metadata, or for reading user-specific preferences such as language or address aliases. Note that if the observed state changes between signature generation and the transaction's inclusion in a block, the delivery-time rendering might differ. If so, the signature will be invalid and the transaction will be rejected.

## Signature and Security

For security, transaction signatures should have three properties:

1. Given the transaction, signatures, and chain state, it must be possible to validate that the signatures matches the transaction, to verify that the signers must have known their respective secret keys.
2. It must be computationally infeasible to find a substantially different transaction for which the given signatures are valid, given the same chain state.
3. The user should be able to give informed consent to the signed data via a simple, secure device with limited display capabilities.

The correctness and security of `SIGN_MODE_TEXTUAL` is guaranteed by demonstrating an inverse function from the rendering to transaction protos. This means that it is impossible for a different protocol buffer message to render to the same text.

## Transaction Hash Malleability

When client software forms a transaction, the "raw" transaction (`TxRaw`) is serialized as a proto and a hash of the resulting byte sequence is computed. This is the `TxHash`, and is used by various services to track the submitted transaction through its lifecycle. Various misbehavior is possible if one can generate a modified transaction with a different `TxHash` but for which the signature still checks out.

`SIGN_MODE_TEXTUAL` prevents this transaction malleability by including the `TxHash` as an expert screen in the rendering.

## SignDoc

The SignDoc for `SIGN_MODE_TEXTUAL` is formed from a data structure like:

```

type Screen struct {
 Title string // possibly size limited to, advised to 64 characters
 Content string // possibly size limited to, advised to 255 characters
 Indent uint8 // size limited to something small like 16 or 32
 Expert bool
}

type SignDocTextual struct {
 Screens []Screen
}

```

We do not plan to use protobuf serialization to form the sequence of bytes that will be transmitted and signed, in order to keep the decoder simple. We will use [CBOR \(RFC 8949\)](#) instead. The encoding is defined by the following CDDL ([RFC 8610](#)):

```

;;, CDDL (RFC 8610) Specification of SignDoc for SIGN_MODE_TEXTUAL.
;;, Must be encoded using CBOR deterministic encoding (RFC 8949, section 4.2.1).

;; A Textual document is a struct containing one field: an array of screens.
sign_doc = {
 screens_key: [* screen],
}

;; The key is an integer to keep the encoding small.
screens_key = 1

;; A screen consists of a text string, an indentation, and the expert flag,
;; represented as an integer-keyed map. All entries are optional
;; and MUST be omitted from the encoding if empty, zero, or false.
;; Text defaults to the empty string, indent defaults to zero,
;; and expert defaults to false.
screen = {
 ? title_key: tstr,
 ? content_key: tstr,
 ? indent_key: uint,
 ? expert_key: bool,
}

;; Keys are small integers to keep the encoding small.
title_key = 1
content_key = 2
indent_key = 3
expert_key = 4

```

Defining the sign\_doc as directly an array of screens has also been considered. However, given the possibility of future iterations of this specification, using a single-keyed struct has been chosen over the former proposal, as structs allow for easier backwards-compatibility.

## Details

In the examples that follow, screens will be shown as lines of text, indentation is indicated with a leading '>', and expert screens are marked with a leading \* .

## Encoding of the Transaction Envelope

We define "transaction envelope" as all data in a transaction that is not in the `TxBody.Messages` field.

Transaction envelope includes fee, signer infos and memo, but don't include `Msg`s. // denotes comments and are not shown on the Ledger device.

```
Chain ID: <string>
Account number: <uint64>
Sequence: <uint64>
Address: <string>
*Public Key: <Any>
This transaction has <int> Message(s) // Pluralize "Message"
only when int>1
> Message (<int>/<int>): <Any> // See value renderers for
Any rendering.
End of Message
Memo: <string> // Skipped if no memo set.
Fee: <coins> // See value renderers for
coins rendering.
*Fee payer: <string> // Skipped if no fee_payer
set.
*Fee granter: <string> // Skipped if no
fee_granter set.
Tip: <coins> // Skipped if no tip.
Tipper: <string>
*Gas Limit: <uint64>
*Timeout Height: <uint64> // Skipped if no
timeout_height set.
*Other signer: <int> SignerInfo // Skipped if the
transaction only has 1 signer.
*> Other signer (<int>/<int>): <SignerInfo>
*End of other signers
*Extension options: <int> Any: // Skipped if no body
extension options
*> Extension options (<int>/<int>): <Any>
*End of extension options
*Non critical extension options: <int> Any: // Skipped if no body non
critical extension options
*> Non critical extension options (<int>/<int>): <Any>
*End of Non critical extension options
*Hash of raw bytes: <hex_string> // Hex encoding of bytes
defined, to prevent tx hash malleability.
```

## Encoding of the Transaction Body

Transaction Body is the `Tx.TxBody.Messages` field, which is an array of `Any`s, where each `Any` packs a `sdk.Msg`. Since `sdk.Msg`s are widely used, they have a slightly different encoding than usual array of `Any`s (`Protobuf: repeated google.protobuf.Any`) described in Annex 1.

```

This transaction has <int> message: // Optional 's' for "message" if there's is >1
sdk.Msgs.

// For each Msg, print the following 2 lines:
Msg (<int>/<int>): <string> // E.g. Msg (1/2): bank v1beta1 send coins
<value rendering of Msg struct>
End of transaction messages

```

## Example

Given the following Protobuf message:

```

message Grant {
 google.protobuf.Any authorization = 1 [(cosmos_proto.accepts_interface) =
"cosmos.authz.v1beta1.Authorization"];
 google.protobuf.Timestamp expiration = 2 [(gogoproto.stdtime) = true,
(gogoproto.nullable) = false];
}

message MsgGrant {
 option (cosmos.msg.v1.signer) = "granter";

 string granter = 1 [(cosmos_proto.scalar) = "cosmos.AddressString"];
 string grantee = 2 [(cosmos_proto.scalar) = "cosmos.AddressString"];
}

```

and a transaction containing 1 such `sdk.Msg`, we get the following encoding:

```

This transaction has 1 message:
Msg (1/1): authz v1beta1 grant
Granter: cosmoslabc...def
Grantee: cosmosighi...jkl
End of transaction messages

```

## Custom `Msg` Renderers

Application developers may choose to not follow default renderer value output for their own `Msg`s. In this case, they can implement their own custom `Msg` renderer. This is similar to [EIP4430](#), where the smart contract developer chooses the description string to be shown to the end user.

This is done by setting the `cosmos.msg.textual.v1.expert_custom_renderer` Protobuf option to a non-empty string. This option CAN ONLY be set on a Protobuf message representing transaction message object (implementing `sdk.Msg` interface).

```

message MsgFooBar {
 // Optional comments to describe in human-readable language the formatting
 // rules of the custom renderer.
 option (cosmos.msg.textual.v1.expert_custom_renderer) = "<unique algorithm
identifier>";

```

```
// proto fields
}
```

When this option is set on a `Msg`, a registered function will transform the `Msg` into an array of one or more strings, which MAY use the key/value format (described in point #3) with the expert field prefix (described in point #5) and arbitrary indentation (point #6). These strings MAY be rendered from a `Msg` field using a default value renderer, or they may be generated from several fields using custom logic.

The `<unique algorithm identifier>` is a string convention chosen by the application developer and is used to identify the custom `Msg` renderer. For example, the documentation or specification of this custom algorithm can reference this identifier. This identifier CAN have a versioned suffix (e.g. `_v1`) to adapt for future changes (which would be consensus-breaking). We also recommend adding Protobuf comments to describe in human language the custom logic used.

Moreover, the renderer must provide 2 functions: one for formatting from Protobuf to string, and one for parsing string to Protobuf. These 2 functions are provided by the application developer. To satisfy point #1, the parse function MUST be the inverse of the formatting function. This property will not be checked by the SDK at runtime. However, we strongly recommend the application developer to include a comprehensive suite in their app repo to test invertibility, as to not introduce security bugs.

### Require signing over the `TxBody` and `AuthInfo` raw bytes

Recall that the transaction bytes merkleized on chain are the Protobuf binary serialization of `TxRaw`, which contains the `body_bytes` and `auth_info_bytes`. Moreover, the transaction hash is defined as the SHA256 hash of the `TxRaw` bytes. We require that the user signs over these bytes in `SIGN_MODE_TEXTUAL`, more specifically over the following string:

```
*Hash of raw bytes: <HEX(sha256(len(body_bytes) ++ body_bytes ++ len(auth_info_bytes)
++ auth_info_bytes))>
```

where:

- `++` denotes concatenation,
- `HEX` is the hexadecimal representation of the bytes, all in capital letters, no `0x` prefix,
- and `len()` is encoded as a Big-Endian uint64.

This is to prevent transaction hash malleability. The point #1 about invertibility assures that transaction `body` and `auth_info` values are not malleable, but the transaction hash still might be malleable with point #1 only, because the `SIGN_MODE_TEXTUAL` strings don't follow the byte ordering defined in `body_bytes` and `auth_info_bytes`. Without this hash, a malicious validator or exchange could intercept a transaction, modify its transaction hash *after* the user signed it using `SIGN_MODE_TEXTUAL` (by tweaking the byte ordering inside `body_bytes` or `auth_info_bytes`), and then submit it to Tendermint.

By including this hash in the `SIGN_MODE_TEXTUAL` signing payload, we keep the same level of guarantees as [SIGN MODE DIRECT](#).

These bytes are only shown in expert mode, hence the leading `*`.

## Updates to the current specification

The current specification is not set in stone, and future iterations are to be expected. We distinguish two categories of updates to this specification:

1. Updates that require changes of the hardware device embedded application.
2. Updates that only modify the envelope and the value renderers.

Updates in the 1st category include changes of the `Screen` struct or its corresponding CBOR encoding.

This type of updates require a modification of the hardware signer application, to be able to decode and parse the new types. Backwards-compatibility must also be guaranteed, so that the new hardware application works with existing versions of the SDK. These updates require the coordination of multiple parties: SDK developers, hardware application developers (currently: Zondax), and client-side developers (e.g. CosmJS). Furthermore, a new submission of the hardware device application may be necessary, which, depending on the vendor, can take some time. As such, we recommend to avoid this type of updates as much as possible.

Updates in the 2nd category include changes to any of the value renderers or to the transaction envelope. For example, the ordering of fields in the envelope can be swapped, or the timestamp formatting can be modified. Since `SIGN_MODE_TEXTUAL` sends `Screen`s to the hardware device, this type of change do not need a hardware wallet application update. They are however state-machine-breaking, and must be documented as such. They require the coordination of SDK developers with client-side developers (e.g. CosmJS), so that the updates are released on both sides close to each other in time.

We define a spec version, which is an integer that must be incremented on each update of either category. This spec version will be exposed by the SDK's implementation, and can be communicated to clients. For example, SDK v0.50 might use the spec version 1, and SDK v0.51 might use 2; thanks to this versioning, clients can know how to craft `SIGN_MODE_TEXTUAL` transactions based on the target SDK version.

The current spec version is defined in the "Status" section, on the top of this document. It is initialized to 0 to allow flexibility in choosing how to define future versions, as it would allow adding a field either in the `SignDoc Go` struct or in Protobuf in a backwards-compatible way.

## Additional Formatting by the Hardware Device

See [annex 2](#).

## Examples

1. A minimal `MsgSend`: [see transaction](#).
2. A transaction with a bit of everything: [see transaction](#).

The examples below are stored in a JSON file with the following fields:

- `proto` : the representation of the transaction in ProtoJSON,
- `screens` : the transaction rendered into `SIGN_MODE_TEXTUAL` screens,
- `cbor` : the sign bytes of the transaction, which is the CBOR encoding of the screens.

## Consequences

### Backwards Compatibility

`SIGN_MODE_TEXTUAL` is purely additive, and doesn't break any backwards compatibility with other sign modes.

### Positive

- Human-friendly way of signing in hardware devices.

- Once SIGN\_MODE\_TEXTUAL is shipped, SIGN\_MODE\_LEGACY\_AMINO\_JSON can be deprecated and removed. On the longer term, once the ecosystem has totally migrated, Amino can be totally removed.

## Negative

- Some fields are still encoded in non-human-readable ways, such as public keys in hexadecimal.
- New ledger app needs to be released, still unclear

## Neutral

- If the transaction is complex, the string array can be arbitrarily long, and some users might just skip some screens and blind sign.

## Further Discussions

- Some details on value renderers need to be polished, see [Annex 1](#).
- Are ledger apps able to support both SIGN\_MODE\_LEGACY\_AMINO\_JSON and SIGN\_MODE\_TEXTUAL at the same time?
- Open question: should we add a Protobuf field option to allow app developers to overwrite the textual representation of certain Protobuf fields and message? This would be similar to Ethereum's [EIP4430](#), where the contract developer decides on the textual representation.
- Internationalization.

## References

- [Annex 1](#)
- Initial discussion: <https://github.com/cosmos/cosmos-sdk/issues/6513>
- Living document used in the working group: <https://hackmd.io/fsZAO-TfT0CKmLDtfMcKeA?both>
- Working group meeting notes: [https://hackmd.io/7RkGfv\\_rQAAzZigUYhcXw](https://hackmd.io/7RkGfv_rQAAzZigUYhcXw)
- Ethereum's "Described Transactions" <https://github.com/ethereum/EIPs/pull/4430>

# ADR 053: Go Module Refactoring

## Changelog

- 2022-04-27: First Draft

## Status

PROPOSED

## Abstract

The current SDK is built as a single monolithic go module. This ADR describes how we refactor the SDK into smaller independently versioned go modules for ease of maintenance.

## Context

Go modules impose certain requirements on software projects with respect to stable version numbers (anything above 0.x) in that [any API breaking changes necessitate a major version](#) increase which technically creates a new go module (with a v2, v3, etc. suffix).

[Keeping modules API compatible](#) in this way requires a fair amount of fair thought and discipline.

The Cosmos SDK is a fairly large project which originated before go modules came into existence and has always been under a v0.x release even though it has been used in production for years now, not because it isn't production quality software, but rather because the API compatibility guarantees required by go modules are fairly complex to adhere to with such a large project. Up to now, it has generally been deemed more important to be able to break the API if needed rather than require all users update all package import paths to accommodate breaking changes causing v2, v3, etc. releases. This is in addition to the other complexities related to protobuf generated code that will be addressed in a separate ADR.

Nevertheless, the desire for semantic versioning has been [strong in the community](#) and the single go module release process has made it very hard to release small changes to isolated features in a timely manner. Release cycles often exceed six months which means small improvements done in a day or two get bottle-necked by everything else in the monolithic release cycle.

## Decision

To improve the current situation, the SDK is being refactored into multiple go modules within the current repository. There has been a [fair amount of debate](#) as to how to do this, with some developers arguing for larger vs smaller module scopes. There are pros and cons to both approaches (which will be discussed below in the [Consequences](#) section), but the approach being adopted is the following:

- a go module should generally be scoped to a specific coherent set of functionality (such as math, errors, store, etc.)
- when code is removed from the core SDK and moved to a new module path, every effort should be made to avoid API breaking changes in the existing code using aliases and wrapper types (as done in <https://github.com/cosmos/cosmos-sdk/pull/10779> and <https://github.com/cosmos/cosmos-sdk/pull/11788>)
- new go modules should be moved to a standalone domain (`cosmossdk.io`) before being tagged as `v1.0.0` to accommodate the possibility that they may be better served by a standalone repository in the future
- all go modules should follow the guidelines in <https://go.dev/blog/module-compatibility> before `v1.0.0` is tagged and should make use of `internal` packages to limit the exposed API surface
- the new go module's API may deviate from the existing code where there are clear improvements to be made or to remove legacy dependencies (for instance on amino or gogo proto), as long the old package attempts to avoid API breakage with aliases and wrappers
- care should be taken when simply trying to turn an existing package into a new go module: <https://github.com/golang/go/wiki/Modules#is-it-possible-to-add-a-module-to-a-multi-module-repository>. In general, it seems safer to just create a new module path (appending v2, v3, etc. if necessary), rather than trying to make an old package a new module.

## Consequences

### Backwards Compatibility

If the above guidelines are followed to use aliases or wrapper types pointing in existing APIs that point back to the new go modules, there should be no or very limited breaking changes to existing APIs.

## Positive

- standalone pieces of software will reach `v1.0.0` sooner
- new features to specific functionality will be released sooner

## Negative

- there will be more go module versions to update in the SDK itself and per-project, although most of these will hopefully be indirect

## Neutral

## Further Discussions

Further discussions are occurring in primarily in <https://github.com/cosmos/cosmos-sdk/discussions/10582> and within the Cosmos SDK Framework Working Group.

## References

- <https://go.dev/doc/modules/release-workflow>
- <https://go.dev/blog/module-compatibility>
- <https://github.com/cosmos/cosmos-sdk/discussions/10162>
- <https://github.com/cosmos/cosmos-sdk/discussions/10582>
- <https://github.com/cosmos/cosmos-sdk/pull/10779>
- <https://github.com/cosmos/cosmos-sdk/pull/11788>

# ADR 054: Semver Compatible SDK Modules

## Changelog

- 2022-04-27: First draft

## Status

DRAFT

## Abstract

In order to move the Cosmos SDK to a system of decoupled semantically versioned modules which can be composed in different combinations (ex. staking v3 with bank v1 and distribution v2), we need to reassess how we organize the API surface of modules to avoid problems with go semantic import versioning and circular dependencies. This ADR explores various approaches we can take to addressing these issues.

## Context

There has been [a fair amount of desire](#) in the community for semantic versioning in the SDK and there has been significant movement to splitting SDK modules into [standalone go modules](#). Both of these will ideally allow the ecosystem to move faster because we won't be waiting for all dependencies to update synchronously. For instance, we could have 3 versions of the core SDK compatible with the latest 2 releases of CosmWasm as well as 4 different versions of staking . This sort of setup would allow early adopters to aggressively integrate new versions, while allowing more conservative users to be selective about which versions they're ready for.

In order to achieve this, we need to solve the following problems:

1. because of the way [go semantic import versioning](#) (SIV) works, moving to SIV naively will actually make it harder to achieve these goals
2. circular dependencies between modules need to be broken to actually release many modules in the SDK independently
3. pernicious minor version incompatibilities introduced through correctly [evolving protobuf schemas](#) without correct [unknown field filtering](#)

Note that all the following discussion assumes that the proto file versioning and state machine versioning of a module are distinct in that:

- proto files are maintained in a non-breaking way (using something like [buf breaking](#) to ensure all changes are backwards compatible)
- proto file versions get bumped much less frequently, i.e. we might maintain `cosmos.bank.v1` through many versions of the bank module state machine
- state machine breaking changes are more common and ideally this is what we'd want to semantically version with go modules, ex. `x/bank/v2`, `x/bank/v3`, etc.

## Problem 1: Semantic Import Versioning Compatibility

Consider we have a module `foo` which defines the following `MsgDoSomething` and that we've released its state machine in go module `example.com/foo`:

```
package foo.v1;

message MsgDoSomething {
 string sender = 1;
 uint64 amount = 2;
}

service Msg {
 DoSomething(MsgDoSomething) returns (MsgDoSomethingResponse);
}
```

Now consider that we make a revision to this module and add a new `condition` field to `MsgDoSomething` and also add a new validation rule on `amount` requiring it to be non-zero, and that following go semantic versioning we release the next state machine version of `foo` as `example.com/foo/v2`.

```
// Revision 1
package foo.v1;

message MsgDoSomething {
 string sender = 1;

 // amount must be a non-zero integer.
 uint64 amount = 2;

 // condition is an optional condition on doing the thing.
 //
 // Since: Revision 1
```

```
Condition condition = 3;
}
```

Approaching this naively, we would generate the protobuf types for the initial version of `foo` in `example.com/foo/types` and we would generate the protobuf types for the second version in `example.com/foo/v2/types`.

Now let's say we have a module `bar` which talks to `foo` using this keeper interface which `foo` provides:

```
type FooKeeper interface {
 DoSomething(MsgDoSomething) error
}
```

### Scenario A: Backward Compatibility: Newer Foo, Older Bar

Imagine we have a chain which uses both `foo` and `bar` and wants to upgrade to `foo/v2`, but the `bar` module has not upgraded to `foo/v2`.

In this case, the chain will not be able to upgrade to `foo/v2` until `bar` has upgraded its references to `example.com/foo/types.MsgDoSomething` to `example.com/foo/v2/types.MsgDoSomething`.

Even if `bar`'s usage of `MsgDoSomething` has not changed at all, the upgrade will be impossible without this change because `example.com/foo/types.MsgDoSomething` and `example.com/foo/v2/types.MsgDoSomething` are fundamentally different incompatible structs in the go type system.

### Scenario B: Forward Compatibility: Older Foo, Newer Bar

Now let's consider the reverse scenario, where `bar` upgrades to `foo/v2` by changing the `MsgDoSomething` reference to `example.com/foo/v2/types.MsgDoSomething` and releases that as `bar/v2` with some other changes that a chain wants. The chain, however, has decided that it thinks the changes in `foo/v2` are too risky and that it'd prefer to stay on the initial version of `foo`.

In this scenario, it is impossible to upgrade to `bar/v2` without upgrading to `foo/v2` even if `bar/v2` would have worked 100% fine with `foo` other than changing the import path to `MsgDoSomething` (meaning that `bar/v2` doesn't actually use any new features of `foo/v2`).

Now because of the way go semantic import versioning works, we are locked into either using `foo` and `bar` OR `foo/v2` and `bar/v2`. We cannot have `foo + bar/v2` OR `foo/v2 + bar`. The go type system doesn't allow this even if both versions of these modules are otherwise compatible with each other.

### Naive Mitigation

A naive approach to fixing this would be to not regenerate the protobuf types in `example.com/foo/v2/types` but instead just update `example.com/foo/types` to reflect the changes needed for `v2` (adding `condition` and requiring `amount` to be non-zero). Then we could release a patch of `example.com/foo/types` with this update and use that for `foo/v2`. But this change is state machine breaking for `v1`. It requires changing the `ValidateBasic` method to reject the case where `amount` is zero, and it adds the `condition` field which should be rejected based on [ADR 020 unknown field filtering](#). So adding these changes as a patch on `v1` is actually incorrect based on semantic

versioning. Chains that want to stay on `v1` of `foo` should not be importing these changes because they are incorrect for `v1`.

## Problem 2: Circular dependencies

None of the above approaches allow `foo` and `bar` to be separate modules if for some reason `foo` and `bar` depend on each other in different ways. For instance, we can't have `foo` import `bar/types` while `bar` imports `foo/types`.

We have several cases of circular module dependencies in the SDK (ex. staking, distribution and slashing) that are legitimate from a state machine perspective. Without separating the API types out somehow, there would be no way to independently semantically version these modules without some other mitigation.

## Problem 3: Handling Minor Version Incompatibilities

Imagine that we solve the first two problems but now have a scenario where `bar/v2` wants the option to use `MsgDoSomething.condition` which only `foo/v2` supports. If `bar/v2` works with `foo v1` and sets `condition` to some non-nil value, then `foo` will silently ignore this field resulting in a silent logic possibly dangerous logic error. If `bar/v2` were able to check whether `foo` was on `v1` or `v2` and dynamically, it could choose to only use `condition` when `foo/v2` is available. Even if `bar/v2` were able to perform this check, however, how do we know that it is always performing the check properly. Without some sort of framework-level [unknown field filtering](#), it is hard to know whether these pernicious hard to detect bugs are getting into our app and a client-server layer such as [ADR 033: Inter-Module Communication](#) may be needed to do this.

# Solutions

## Approach A) Separate API and State Machine Modules

One solution (first proposed in <https://github.com/cosmos/cosmos-sdk/discussions/10582>) is to isolate all protobuf generated code into a separate module from the state machine module. This would mean that we could have state machine go modules `foo` and `foo/v2` which could use a types or API go module say `foo/api`. This `foo/api` go module would be perpetually on `v1.x` and only accept non-breaking changes. This would then allow other modules to be compatible with either `foo` or `foo/v2` as long as the inter-module API only depends on the types in `foo/api`. It would also allow modules `foo` and `bar` to depend on each other in that both of them could depend on `foo/api` and `bar/api` without `foo` directly depending on `bar` and vice versa.

This is similar to the naive mitigation described above except that it separates the types into separate go modules which in and of itself could be used to break circular module dependencies. It has the same problems as the naive solution, otherwise, which we could rectify by:

1. removing all state machine breaking code from the API module (ex. `ValidateBasic` and any other interface methods)
2. embedding the correct file descriptors for unknown field filtering in the binary

## Migrate all interface methods on API types to handlers

To solve 1), we need to remove all interface implementations from generated types and instead use a handler approach which essentially means that given a type `x`, we have some sort of resolver which allows us to resolve interface implementations for that type (ex. `sdk.Msg` or `authz.Authorization`). For example:

```

func (k Keeper) DoSomething(msg MsgDoSomething) error {
 var validateBasicHandler ValidateBasicHandler
 err := k.resolver.Resolve(&validateBasic, msg)
 if err != nil {
 return err
 }

 err = validateBasicHandler.ValidateBasic()
 ...
}

```

In the case of some methods on `sdk.Msg`, we could replace them with declarative annotations. For instance, `GetSigners` can already be replaced by the protobuf annotation `cosmos.msg.v1.signer`. In the future, we may consider some sort of protobuf validation framework (like <https://github.com/bufbuild/protoc-gen-validate> but more Cosmos-specific) to replace `ValidateBasic`.

### Pinned FileDescriptor's

To solve 2), state machine modules must be able to specify what the version of the protobuf files was that they were built against. For instance if the API module for `foo` upgrades to `foo/v2`, the original `foo` module still needs a copy of the original protobuf files it was built with so that ADR 020 unknown field filtering will reject `MsgDoSomething` when `Condition` is set.

The simplest way to do this may be to embed the protobuf `FileDescriptor`'s into the module itself so that these `FileDescriptor`'s are used at runtime rather than the ones that are built into the `foo/api` which may be different. Using [buf build](#), [go embed](#), and a build script we can probably come up with a solution for embedding `FileDescriptor`'s into modules that is fairly straightforward.

### Potential limitations to generated code

One challenge with this approach is that it places heavy restrictions on what can go in API modules and requires that most of this is state machine breaking. All or most of the code in the API module would be generated from protobuf files, so we can probably control this with how code generation is done, but it is a risk to be aware of.

For instance, we do code generation for the ORM that in the future could contain optimizations that are state machine breaking. We would either need to ensure very carefully that the optimizations aren't actually state machine breaking in generated code or separate this generated code out from the API module into the state machine module. Both of these mitigations are potentially viable but the API module approach does require an extra level of care to avoid these sorts of issues.

### Minor Version Incompatibilities

This approach in and of itself does little to address any potential minor version incompatibilities and the requisite [unknown field filtering](#). Likely some sort of client-server routing layer which does this check such as [ADR 033: Inter-Module communication](#) is required to make sure that this is done properly. We could then allow modules to perform a runtime check given a `MsgClient`, ex:

```

func (k Keeper) CallFoo() error {
 if k.interModuleClient.MinorRevision(k.fooMsgClient) >= 2 {
 k.fooMsgClient.DoSomething(&MsgDoSomething{Condition: ...})
 } else {

```

```
 ...
}
```

To do the unknown field filtering itself, the ADR 033 router would need to use the [protoreflect API](#) to ensure that no fields unknown to the receiving module are set. This could result in an undesirable performance hit depending on how complex this logic is.

## Approach B) Changes to Generated Code

An alternate approach to solving the versioning problem is to change how protobuf code is generated and move modules mostly or completely in the direction of inter-module communication as described in [ADR 033](#). In this paradigm, a module could generate all the types it needs internally - including the API types of other modules - and talk to other modules via a client-server boundary. For instance, if `bar` needs to talk to `foo`, it could generate its own version of `MsgDoSomething` as `bar/internal/foo/v1.MsgDoSomething` and just pass this to the inter-module router which would somehow convert it to the version which `foo` needs (ex. `foo/internal.MsgDoSomething`).

Currently, two generated structs for the same protobuf type cannot exist in the same go binary without special build flags (see <https://developers.google.com/protocol-buffers/docs/reference/go/faq#fix-namespace-conflict>). A relatively simple mitigation to this issue would be to set up the protobuf code to not register protobuf types globally if they are generated in an `internal/` package. This will require modules to register their types manually with the app-level level protobuf registry, this is similar to what modules already do with the `InterfaceRegistry` and amino codec.

If modules *only* do ADR 033 message passing then a naive and non-performant solution for converting `bar/internal/foo/v1.MsgDoSomething` to `foo/internal.MsgDoSomething` would be marshaling and unmarshaling in the ADR 033 router. This would break down if we needed to expose protobuf types in `Keeper` interfaces because the whole point is to try to keep these types `internal/` so that we don't end up with all the import version incompatibilities we've described above. However, because of the issue with minor version incompatibilities and the need for [unknown field filtering](#), sticking with the `Keeper` paradigm instead of ADR 033 may be unviable to begin with.

A more performant solution (that could maybe be adapted to work with `Keeper` interfaces) would be to only expose getters and setters for generated types and internally store data in memory buffers which could be passed from one implementation to another in a zero-copy way.

For example, imagine this protobuf API with only getters and setters is exposed for `MsgSend`:

```
type MsgSend interface {
 proto.Message
 GetFromAddress() string
 GetToAddress() string
 GetAmount() []v1beta1.Coin
 SetFromAddress(string)
 SetToAddress(string)
 SetAmount([]v1beta1.Coin)
}

func NewMsgSend() MsgSend { return &msgSendImpl{memoryBuffers: ...} }
```

Under the hood, `MsgSend` could be implemented based on some raw memory buffer in the same way that [Cap'n Proto](#) and [FlatBuffers](#) so that we could convert between one version of `MsgSend` and another without serialization (i.e. zero-copy). This approach would have the added benefits of allowing zero-copy message passing to modules written in other languages such as Rust and accessed through a VM or FFI. It could also make unknown field filtering in inter-module communication simpler if we require that all new fields are added in sequential order, ex. just checking that no field `> 5` is set.

Also, we wouldn't have any issues with state machine breaking code on generated types because all the generated code used in the state machine would actually live in the state machine module itself. Depending on how interface types and protobuf `Any`s are used in other languages, however, it may still be desirable to take the handler approach described in approach A. Either way, types implementing interfaces would still need to be registered with an `InterfaceRegistry` as they are now because there would be no way to retrieve them via the global registry.

In order to simplify access to other modules using ADR 033, a public API module (maybe even one [remotely generated by Buf](#)) could be used by client modules instead of requiring to generate all client types internally.

The big downsides of this approach are that it requires big changes to how people use protobuf types and would be a substantial rewrite of the protobuf code generator. This new generated code, however, could still be made compatible with the [google.github.io/protobuf/reflect/protoreflect](https://google.github.io/protobuf/reflect/protoreflect) API in order to work with all standard golang protobuf tooling.

It is possible that the naive approach of marshaling/unmarshaling in the ADR 033 router is an acceptable intermediate solution if the changes to the code generator are seen as too complex. However, since all modules would likely need to migrate to ADR 033 anyway with this approach, it might be better to do this all at once.

### Approach C) Don't address these issues

If the above solutions are seen as too complex, we can also decide not to do anything explicit to enable better module version compatibility, and break circular dependencies.

In this case, when developers are confronted with the issues described above they can require dependencies to update in sync (what we do now) or attempt some ad-hoc potentially hacky solution.

One approach is to ditch go semantic import versioning (SIV) altogether. Some people have commented that go's SIV (i.e. changing the import path to `foo/v2`, `foo/v3`, etc.) is too restrictive and that it should be optional. The golang maintainers disagree and only officially support semantic import versioning. We could, however, take the contrarian perspective and get more flexibility by using 0.x-based versioning basically forever.

Module version compatibility could then be achieved using go.mod replace directives to pin dependencies to specific compatible 0.x versions. For instance if we knew `foo` 0.2 and 0.3 were both compatible with `bar` 0.3 and 0.4, we could use replace directives in our go.mod to stick to the versions of `foo` and `bar` we want. This would work as long as the authors of `foo` and `bar` avoid incompatible breaking changes between these modules.

Or, if developers choose to use semantic import versioning, they can attempt the naive solution described above and would also need to use special tags and replace directives to make sure that modules are pinned to the correct versions.

Note, however, that all of these ad-hoc approaches, would be vulnerable to the minor version compatibility issues described above unless [unknown field filtering](#) is properly addressed.

### Approach D) Avoid protobuf generated code in public APIs

An alternative approach would be to avoid protobuf generated code in public module APIs. This would help avoid the discrepancy between state machine versions and client API versions at the module to module boundaries. It would mean that we wouldn't do inter-module message passing based on ADR 033, but rather stick to the existing keeper approach and take it one step further by avoiding any protobuf generated code in the keeper interface methods.

Using this approach, our `foo.Keeper.DoSomething` method wouldn't have the generated `MsgDoSomething` struct (which comes from the protobuf API), but instead positional parameters. Then in order for `foo/v2` to support the `foo/v1` keeper it would simply need to implement both the v1 and v2 keeper APIs. The `DoSomething` method in v2 could have the additional `condition` parameter, but this wouldn't be present in v1 at all so there would be no danger of a client accidentally setting this when it isn't available.

So this approach would avoid the challenge around minor version incompatibilities because the existing module keeper API would not get new fields when they are added to protobuf files.

Taking this approach, however, would likely require making all protobuf generated code internal in order to prevent it from leaking into the keeper API. This means we would still need to modify the protobuf code generator to not register `internal/` code with the global registry, and we would still need to manually register protobuf `FileDescriptor`s (this is probably true in all scenarios). It may, however, be possible to avoid needing to refactor interface methods on generated types to handlers.

Also, this approach doesn't address what would be done in scenarios where modules still want to use the message router. Either way, we probably still want a way to pass messages from one module to another router safely even if it's just for use cases like `x/gov`, `x/authz`, `CosmWasm`, etc. That would still require most of the things outlined in approach (B), although we could advise modules to prefer keepers for communicating with other modules.

The biggest downside of this approach is probably that it requires a strict refactoring of keeper interfaces to avoid generated code leaking into the API. This may result in cases where we need to duplicate types that are already defined in proto files and then write methods for converting between the golang and protobuf version. This may end up in a lot of unnecessary boilerplate and that may discourage modules from actually adopting it and achieving effective version compatibility. Approaches (A) and (B), although heavy handed initially, aim to provide a system which once adopted more or less gives the developer version compatibility for free with minimal boilerplate. Approach (D) may not be able to provide such a straightforward system since it requires a golang API to be defined alongside a protobuf API in a way that requires duplication and differing sets of design principles (protobuf APIs encourage additive changes while golang APIs would forbid it).

Other downsides to this approach are:

- no clear roadmap to supporting modules in other languages like Rust
- doesn't get us any closer to proper object capability security (one of the goals of ADR 033)
- ADR 033 needs to be done properly anyway for the set of use cases which do need it

## Decision

The latest **DRAFT** proposal is:

1. we are alignment on adopting [ADR 033](#) not just as an addition to the framework, but as a core replacement to the keeper paradigm entirely.
2. the ADR 033 inter-module router will accommodate any variation of approach (A) or (B) given the following rules: a. if the client type is the same as the server type then pass it directly through, b. if both client and server use the zero-copy generated code wrappers (which still need to be defined), then pass the memory buffers from one wrapper to the other, or c. marshal/unmarshal types between client and server.

This approach will allow for both maximal correctness and enable a clear path to enabling modules within in other languages, possibly executed within a WASM VM.

## Minor API Revisions

To declare minor API revisions of proto files, we propose the following guidelines (which were already documented in [cosmos.app.v1alpha module options](#)):

- proto packages which are revised from their initial version (considered revision 0 ) should include a package
- comment in some .proto file containing the test Revision N at the start of a comment line where N is the current revision number.
- all fields, messages, etc. added in a version beyond the initial revision should add a comment at the start of a comment line of the form Since: Revision N where N is the non-zero revision it was added.

It is advised that there is a 1:1 correspondence between a state machine module and versioned set of proto files which are versioned either as a buf module a go API module or both. If the buf schema registry is used, the version of this buf module should always be 1.N where N corresponds to the package revision. Patch releases should be used when only documentation comments are updated. It is okay to include proto packages named v2 , v3 , etc. in this same 1.N versioned buf module (ex. cosmos.bank.v2 ) as long as all these proto packages consist of a single API intended to be served by a single SDK module.

## Introspecting Minor API Revisions

In order for modules to introspect the minor API revision of peer modules, we propose adding the following method to `cosmossdk.io/core/intermodule.Client` :

```
ServiceRevision(ctx context.Context, serviceName string) uint64
```

Modules could all this using the service name statically generated by the go grpc code generator:

```
intermoduleClient.ServiceRevision(ctx, bankv1beta1.Msg_ServiceDesc.ServiceName)
```

In the future, we may decide to extend the code generator used for protobuf services to add a field to client types which does this check more concisely, ex:

```
package bankv1beta1

type MsgClient interface {
 Send(context.Context, MsgSend) (MsgSendResponse, error)
```

```
 ServiceRevision(context.Context) uint64
}
```

## Unknown Field Filtering

To correctly perform [unknown field filtering](#), the inter-module router can do one of the following:

- use the `protoreflect` API for messages which support that
- for gogo proto messages, marshal and use the existing `codec/unknownproto` code
- for zero-copy messages, do a simple check on the highest set field number (assuming we can require that fields are adding consecutively in increasing order)

## FileDescriptor Registration

Because a single go binary may contain different versions of the same generated protobuf code, we cannot rely on the global protobuf registry to contain the correct `FileDescriptor`s. Because `appconfig` module configuration is itself written in protobuf, we would like to load the `FileDescriptor`s for a module before loading a module itself. So we will provide ways to register `FileDescriptor`s at module registration time before instantiation. We propose the following `cosmossdk.io/core/appmodule.Option` constructors for the various cases of how `FileDescriptor`s may be packaged:

```
package appmodule

// this can be used when we are using google.golang.org/protobuf compatible
generated code
// Ex:
// ProtoFiles(bankv1beta1.File_cosmos_bank_v1beta1_module_proto)
func ProtoFiles(file []protoreflect.FileDescriptor) Option {}

// this can be used when we are using gogo proto generated code.
func GzippedProtoFiles(file [][]byte) Option {}

// this can be used when we are using buf build to generate a pinned file
descriptor
func ProtoImage(protoImage []byte) Option {}
```

This approach allows us to support several ways protobuf files might be generated:

- proto files generated internally to a module (use `ProtoFiles`)
- the API module approach with pinned file descriptors (use `ProtoImage`)
- gogo proto (use `GzippedProtoFiles`)

## Module Dependency Declaration

One risk of ADR 033 is that dependencies are called at runtime which are not present in the loaded set of SDK modules.

Also we want modules to have a way to define a minimum dependency API revision that they require. Therefore, all modules should declare their set of dependencies upfront. These dependencies could be defined when a module is instantiated, but ideally we know what the dependencies are before instantiation and can statically look at an app config and determine whether the set of modules. For example, if `bar` requires `foo` revision `>= 1`, then we should be able to know this when creating an app config with two versions of `bar` and `foo`.

We propose defining these dependencies in the proto options of the module config object itself.

## Interface Registration

We will also need to define how interface methods are defined on types that are serialized as `google.protobuf.Any`'s. In light of the desire to support modules in other languages, we may want to think of solutions that will accommodate other languages such as plugins described briefly in [ADR 033](#).

## Testing

In order to ensure that modules are indeed with multiple versions of their dependencies, we plan to provide specialized unit and integration testing infrastructure that automatically tests multiple versions of dependencies.

### Unit Testing

Unit tests should be conducted inside SDK modules by mocking their dependencies. In a full ADR 033 scenario, this means that all interaction with other modules is done via the inter-module router, so mocking of dependencies means mocking their msg and query server implementations. We will provide both a test runner and fixture to make this streamlined. The key thing that the test runner should do to test compatibility is to test all combinations of dependency API revisions. This can be done by taking the file descriptors for the dependencies, parsing their comments to determine the revisions various elements were added, and then created synthetic file descriptors for each revision by subtracting elements that were added later.

Here is a proposed API for the unit test runner and fixture:

```
package moduletesting

import (
 "context"
 "testing"

 "cosmosdk.io/core/intermodule"
 "cosmosdk.io/depinject"
 "google.golang.org/grpc"
 "google.golang.org/protobuf/proto"
 "google.golang.org/protobuf/reflect/protodesc"
)

type TestFixture interface {
 context.Context
 intermodule.Client // for making calls to the module we're testing
 BeginBlock()
 EndBlock()
}

type UnitTestFixture interface {
 TestFixture
 grpc.ServiceRegistrar // for registering mock service implementations
}

type UnitTestConfig struct {
```

```

ModuleConfig proto.Message // the module's config object
DepinjectConfig depinjected.Config // optional additional depinject
config options
DependencyFileDescriptors []protodesc.FileDescriptorProto // optional dependency
file descriptors to use instead of the global registry
}

// Run runs the test function for all combinations of dependency API revisions.
func (cfg UnitTestConfig) Run(t *testing.T, f func(t *testing.T, f UnitTestFixture))
{
 // ...
}

```

Here is an example for testing bar calling foo which takes advantage of conditional service revisions in the expected mock arguments:

```

func TestBar(t *testing.T) {
 UnitTestConfig{ModuleConfig: &foomodulev1.Module{}}.Run(t, func (t *testing.T, f
moduletesting.UnitTestFixture) {
 ctrl := gomock.NewController(t)
 mockFooMsgServer := foottestutil.NewMockMsgServer()
 foov1.RegisterMsgServer(f, mockFooMsgServer)
 barMsgClient := barv1.NewMsgClient(f)
 if f.ServiceRevision(foov1.Msg_ServiceDesc.ServiceName) >= 1 {
 mockFooMsgServer.EXPECT().DoSomething(gomock.Any(),
&foov1.MsgDoSomething{
 ...,
 Condition: ..., // condition is expected in revision >= 1
 }).Return(&foov1.MsgDoSomethingResponse{}, nil)
 } else {
 mockFooMsgServer.EXPECT().DoSomething(gomock.Any(),
&foov1.MsgDoSomething{...}).Return(&foov1.MsgDoSomethingResponse{}, nil)
 }
 res, err := barMsgClient.CallFoo(f, &MsgCallFoo{})
 ...
})
}

```

The unit test runner would make sure that no dependency mocks return arguments which are invalid for the service revision being tested to ensure that modules don't incorrectly depend on functionality not present in a given revision.

## Integration Testing

An integration test runner and fixture would also be provided which instead of using mocks would test actual module dependencies in various combinations. Here is the proposed API:

```

type IntegrationTestFixture interface {
 TestFixture
}

```

```

type IntegrationTestConfig struct {
 ModuleConfig proto.Message // the module's config object
 DependencyMatrix map[string][]proto.Message // all the dependent module configs
}

// Run runs the test function for all combinations of dependency modules.
func (cfg IntegrationTestConfig) Run(t *testing.T, f func (t *testing.T, f
IntegrationTestFixture)) {
 // ...
}

```

And here is an example with foo and bar:

```

func TestBarIntegration(t *testing.T) {
 IntegrationTestConfig{
 ModuleConfig: &barmodulev1.Module{},
 DependencyMatrix: map[string][]proto.Message{
 "runtime": []proto.Message{ // test against two versions of runtime
 &runtimev1.Module{},
 &runtimev2.Module{},
 },
 "foo": []proto.Message{ // test against three versions of foo
 &foomodulev1.Module{},
 &foomodulev2.Module{},
 &foomodulev3.Module{},
 }
 }
 }.Run(t, func (t *testing.T, f moduletesting.IntegrationTestFixture) {
 barMsgClient := barv1.NewMsgClient(f)
 res, err := barMsgClient.CallFoo(f, &MsgCallFoo{})
 ...
 })
}

```

Unlike unit tests, integration tests actually pull in other module dependencies. So that modules can be written without direct dependencies on other modules and because golang has no concept of development dependencies, integration tests should be written in separate go modules, ex.

[example.com/bar/v2/test](http://example.com/bar/v2/test). Because this paradigm uses go semantic versioning, it is possible to build a single go module which imports 3 versions of bar and 2 versions of runtime and can test these all together in the six various combinations of dependencies.

## Consequences

### Backwards Compatibility

Modules which migrate fully to ADR 033 will not be compatible with existing modules which use the keeper paradigm. As a temporary workaround we may create some wrapper types that emulate the current keeper interface to minimize the migration overhead.

### Positive

- we will be able to deliver interoperable semantically versioned modules which should dramatically increase the ability of the Cosmos SDK ecosystem to iterate on new features
- it will be possible to write Cosmos SDK modules in other languages in the near future

## Negative

- all modules will need to be refactored somewhat dramatically

## Neutral

- the `cosmossdk.io/core/appconfig` framework will play a more central role in terms of how modules are defined, this is likely generally a good thing but does mean additional changes for users wanting to stick to the pre-depinject way of wiring up modules
- `depinject` is somewhat less needed or maybe even obviated because of the full ADR 033 approach. If we adopt the core API proposed in <https://github.com/cosmos/cosmos-sdk/pull/12239>, then a module would probably always instantiate itself with a method `ProvideModule(appmodule.Service) (appmodule.AppModule, error)`. There is no complex wiring of keeper dependencies in this scenario and dependency injection may not have as much of (or any) use case.

## Further Discussions

The decision described above is considered in draft mode and is pending final buy-in from the team and key stakeholders. Key outstanding discussions if we do adopt that direction are:

- how do module clients introspect dependency module API revisions
- how do modules determine a minor dependency module API revision requirement
- how do modules appropriately test compatibility with different dependency versions
- how to register and resolve interface implementations
- how do modules register their protobuf file descriptors depending on the approach they take to generated code (the API module approach may still be viable as a supported strategy and would need pinned file descriptors)

## References

- <https://github.com/cosmos/cosmos-sdk/discussions/10162>
- <https://github.com/cosmos/cosmos-sdk/discussions/10582>
- <https://github.com/cosmos/cosmos-sdk/discussions/10368>
- <https://github.com/cosmos/cosmos-sdk/pull/11340>
- <https://github.com/cosmos/cosmos-sdk/issues/11899>
- [ADR 020](#)
- [ADR 033](#)

## ADR 055: ORM

### Changelog

- 2022-04-27: First draft

### Status

ACCEPTED Implemented

## Abstract

In order to make it easier for developers to build Cosmos SDK modules and for clients to query, index and verify proofs against state data, we have implemented an ORM (object-relational mapping) layer for the Cosmos SDK.

## Context

Historically modules in the Cosmos SDK have always used the key-value store directly and created various handwritten functions for managing key format as well as constructing secondary indexes. This consumes a significant amount of time when building a module and is error-prone. Because key formats are non-standard, sometimes poorly documented, and subject to change, it is hard for clients to generically index, query and verify merkle proofs against state data.

The known first instance of an "ORM" in the Cosmos ecosystem was in [weave](#). A later version was built for [regen-ledger](#) for use in the group module and later [ported to the SDK](#) just for that purpose.

While these earlier designs made it significantly easier to write state machines, they still required a lot of manual configuration, didn't expose state format directly to clients, and were limited in their support of different types of index keys, composite keys, and range queries.

Discussions about the design continued in <https://github.com/cosmos/cosmos-sdk/discussions/9156> and more sophisticated proofs of concept were created in <https://github.com/allinbits/cosmos-sdk-poc/tree/master/runtime/orm> and <https://github.com/cosmos/cosmos-sdk/pull/10454>.

## Decision

These prior efforts culminated in the creation of the Cosmos SDK `orm` go module which uses protobuf annotations for specifying ORM table definitions. This ORM is based on the new [google.github.io/protobuf/reflect/protoreflect](https://google.github.io/protobuf/reflect/protoreflect) API and supports:

- sorted indexes for all simple protobuf types (except `bytes`, `enum`, `float`, `double`) as well as `Timestamp` and `Duration`
- unsorted `bytes` and `enum` indexes
- composite primary and secondary keys
- unique indexes
- auto-incrementing `uint64` primary keys
- complex prefix and range queries
- paginated queries
- complete logical decoding of KV-store data

Almost all the information needed to decode state directly is specified in .proto files. Each table definition specifies an ID which is unique per .proto file and each index within a table is unique within that table. Clients then only need to know the name of a module and the prefix ORM data for a specific .proto file within that module in order to decode state data directly. This additional information will be exposed directly through app configs which will be explained in a future ADR related to app wiring.

The ORM makes optimizations around storage space by not repeating values in the primary key in the key value when storing primary key records. For example, if the object `{"a":0,"b":1}` has the primary key `a`, it will be stored in the key value store as `Key: '0', Value: {"b":1}` (with more efficient protobuf binary encoding). Also, the generated code from <https://github.com/cosmos/cosmos-proto> does

optimizations around the `google.golang.org/protobuf/reflect/protoreflect` API to improve performance.

A code generator is included with the ORM which creates type safe wrappers around the ORM's dynamic `Table` implementation and is the recommended way for modules to use the ORM.

The ORM tests provide a simplified bank module demonstration which illustrates:

- [ORM proto options](#)
- [Generated Code](#)
- [Example Usage in a Module Keeper](#)

## Consequences

### Backwards Compatibility

State machine code that adopts the ORM will need migrations as the state layout is generally backwards incompatible. These state machines will also need to migrate to <https://github.com/cosmos/cosmos-proto> at least for state data.

### Positive

- easier to build modules
- easier to add secondary indexes to state
- possible to write a generic indexer for ORM state
- easier to write clients that do state proofs
- possible to automatically write query layers rather than needing to manually implement gRPC queries

### Negative

- worse performance than handwritten keys (for now). See [Further Discussions](#) for potential improvements

### Neutral

## Further Discussions

Further discussions will happen within the Cosmos SDK Framework Working Group. Current planned and ongoing work includes:

- automatically generate client-facing query layer
- client-side query libraries that transparently verify light client proofs
- index ORM data to SQL databases
- improve performance by:
  - optimizing existing reflection based code to avoid unnecessary gets when doing deletes & updates of simple tables
  - more sophisticated code generation such as making fast path reflection even faster (avoiding `switch` statements), or even fully generating code that equals handwritten performance

## References

- <https://github.com/iov-one/weave/tree/master/orm>.

- <https://github.com/regen-network/regen-ledger/tree/157181f955823149e1825263a317ad8e16096da4/orm>
- <https://github.com/cosmos/cosmos-sdk/tree/35d3312c3be306591fcba39892223f1244c8d108/x/group/internal/orm>
- <https://github.com/cosmos/cosmos-sdk/discussions/9156>
- <https://github.com/allinbits/cosmos-sdk-poc/tree/master/runtime/orm>
- <https://github.com/cosmos/cosmos-sdk/pull/10454>

# ADR 057: App Wiring

## Changelog

- 2022-05-04: Initial Draft
- 2022-08-19: Updates

## Status

PROPOSED Implemented

## Abstract

In order to make it easier to build Cosmos SDK modules and apps, we propose a new app wiring system based on dependency injection and declarative app configurations to replace the current `app.go` code.

## Context

A number of factors have made the SDK and SDK apps in their current state hard to maintain. A symptom of the current state of complexity is `simapp/app.go` which contains almost 100 lines of imports and is otherwise over 600 lines of mostly boilerplate code that is generally copied to each new project. (Not to mention the additional boilerplate which gets copied in `simapp/simd`.)

The large amount of boilerplate needed to bootstrap an app has made it hard to release independently versioned go modules for Cosmos SDK modules as described in [ADR 053: Go Module Refactoring](#).

In addition to being very verbose and repetitive, `app.go` also exposes a large surface area for breaking changes as most modules instantiate themselves with positional parameters which forces breaking changes anytime a new parameter (even an optional one) is needed.

Several attempts were made to improve the current situation including [ADR 033: Internal-Module Communication](#) and [a proof-of-concept of a new SDK](#). The discussions around these designs led to the current solution described here.

## Decision

In order to improve the current situation, a new "app wiring" paradigm has been designed to replace `app.go` which involves:

- declaration configuration of the modules in an app which can be serialized to JSON or YAML
- a dependency-injection (DI) framework for instantiating apps from the that configuration

## Dependency Injection

When examining the code in `app.go` most of the code simply instantiates modules with dependencies provided either by the framework (such as store keys) or by other modules (such as keepers). It is generally pretty obvious given the context what the correct dependencies actually should be, so dependency-injection is an obvious solution. Rather than making developers manually resolve dependencies, a module will tell the DI container what dependency it needs and the container will figure out how to provide it.

We explored several existing DI solutions in golang and felt that the reflection-based approach in [uber/dig](#) was closest to what we needed but not quite there. Assessing what we needed for the SDK, we designed and built the Cosmos SDK [depinject module](#), which has the following features:

- dependency resolution and provision through functional constructors, ex: `func (need SomeDep) (AnotherDep, error)`
- dependency injection `In` and `Out` structs which support `optional` dependencies
- grouped-dependencies (many-per-container) through the `ManyPerContainerType` tag interface
- module-scoped dependencies via `ModuleKey`s (where each module gets a unique dependency)
- one-per-module dependencies through the `OnePerModuleType` tag interface
- sophisticated debugging information and container visualization via GraphViz

Here are some examples of how these would be used in an SDK module:

- `StoreKey` could be a module-scoped dependency which is unique per module
- a module's `AppModule` instance (or the equivalent) could be a `OnePerModuleType`
- CLI commands could be provided with `ManyPerContainerType`s

Note that even though dependency resolution is dynamic and based on reflection, which could be considered a pitfall of this approach, the entire dependency graph should be resolved immediately on app startup and only gets resolved once (except in the case of dynamic config reloading which is a separate topic). This means that if there are any errors in the dependency graph, they will get reported immediately on startup so this approach is only slightly worse than fully static resolution in terms of error reporting and much better in terms of code complexity.

## Declarative App Config

In order to compose modules into an app, a declarative app configuration will be used. This configuration is based off of protobuf and its basic structure is very simple:

```
package cosmos.app.v1;

message Config {
 repeated ModuleConfig modules = 1;
}

message ModuleConfig {
 string name = 1;
 google.protobuf.Any config = 2;
}
```

(See also <https://github.com/cosmos/cosmos-sdk/blob/6e18f582bf69e3926a1e22a6de3c35ea327aadce/proto/cosmos/app/v1alpha1/config.proto>)

The configuration for every module is itself a protobuf message and modules will be identified and loaded based on the protobuf type URL of their config object (ex. `cosmos.bank.module.v1.Module`). Modules

are given a unique short `name` to share resources across different versions of the same module which might have a different protobuf package versions (ex. `cosmos.bank.module.v2.Module`). All module config objects should define the `cosmos.app.v1alpha1.module` descriptor option which will provide additional useful metadata for the framework and which can also be indexed in module registries.

An example app config in YAML might look like this:

```
modules:
- name: baseapp
 config:
 "@type": cosmos.baseapp.module.v1.Module
 begin_blockers: [staking, auth, bank]
 end_blockers: [bank, auth, staking]
 init_genesis: [bank, auth, staking]
- name: auth
 config:
 "@type": cosmos.auth.module.v1.Module
 bech32_prefix: "foo"
- name: bank
 config:
 "@type": cosmos.bank.module.v1.Module
- name: staking
 config:
 "@type": cosmos.staking.module.v1.Module
```

In the above example, there is a hypothetical `baseapp` module which contains the information around ordering of begin blockers, end blockers, and init genesis. Rather than lifting these concerns up to the module config layer, they are themselves handled by a module which could allow a convenient way of swapping out different versions of baseapp (for instance to target different versions of tendermint), without needing to change the rest of the config. The `baseapp` module would then provide to the server framework (which sort of sits outside the ABCI app) an instance of `abci.Application`.

In this model, an app is *modules all the way down* and the dependency injection/app config layer is very much protocol-agnostic and can adapt to even major breaking changes at the protocol layer.

## Module & Protobuf Registration

In order for the two components of dependency injection and declarative configuration to work together as described, we need a way for modules to actually register themselves and provide dependencies to the container.

One additional complexity that needs to be handled at this layer is protobuf registry initialization. Recall that in both the current SDK `codec` and the proposed [ADR 054: Protobuf Semver Compatible Codegen](#), protobuf types need to be explicitly registered. Given that the app config itself is based on protobuf and uses protobuf `Any` types, protobuf registration needs to happen before the app config itself can be decoded. Because we don't know which protobuf `Any` types will be needed a priori and modules themselves define those types, we need to decode the app config in separate phases:

1. parse app config JSON/YAML as raw JSON and collect required module type URLs (without doing proto JSON decoding)
2. build a [protobuf type registry](#) based on file descriptors and types provided by each required module

### 3. decode the app config as proto JSON using the protobuf type registry

Because in [ADR 054: Protobuf Semver Compatible Codegen](#), each module might use `internal` generated code which is not registered with the global protobuf registry, this code should provide an alternate way to register protobuf types with a type registry. In the same way that `.pb.go` files currently have a `var File_foo_proto protoreflect.FileDescriptor` for the file `foo.proto`, generated code should have a new member `var Types_foo_proto TypeInfo` where `TypeInfo` is an interface or struct with all the necessary info to register both the protobuf generated types and file descriptor.

So a module must provide dependency injection providers and protobuf types, and takes as input its module config object which uniquely identifies the module based on its type URL.

With this in mind, we define a global module register which allows module implementations to register themselves with the following API:

```
// Register registers a module with the provided type name (ex.
cosmos.bank.module.v1.Module)
// and the provided options.
func Register(configTypeName protoreflect.FullName, option ...Option) { ... }

type Option { /* private methods */ }

// Provide registers dependency injection provider functions which work with the
// cosmos-sdk container module. These functions can also accept an additional
// parameter for the module's config object.
func Provide(providers ...interface{}) Option { ... }

// Types registers protobuf TypeInfo's with the protobuf registry.
func Types(types ...TypeInfo) Option { ... }
```

Ex:

```
func init() {
 appmodule.Register("cosmos.bank.module.v1.Module",
 appmodule.Types(
 types.Types_tx_proto,
 types.Types_query_proto,
 types.Types_types_proto,
),
 appmodule.Provide(
 provideBankModule,
)
)
}

type Inputs struct {
 container.In

 AuthKeeper auth.Keeper
 DB ormdb.ModuleDB
}
```

```

type Outputs struct {
 Keeper bank.Keeper
 AppModule appmodule.AppModule
}

func ProvideBankModule(config *bankmodulev1.Module, Inputs) (Outputs, error) { ... }

```

Note that in this module, a module configuration object *cannot* register different dependency providers at runtime based on the configuration. This is intentional because it allows us to know globally which modules provide which dependencies, and it will also allow us to do code generation of the whole app initialization. This can help us figure out issues with missing dependencies in an app config if the needed modules are loaded at runtime. In cases where required modules are not loaded at runtime, it may be possible to guide users to the correct module if through a global Cosmos SDK module registry.

The `*appmodule.Handler` type referenced above is a replacement for the legacy `AppModule` framework, and described in [ADR 063: Core Module API](#).

## New `app.go`

With this setup, `app.go` might now look something like this:

```

package main

import (
 // Each go package which registers a module must be imported just for side-
 // effects
 // so that module implementations are registered.
 _ "github.com/cosmos/cosmos-sdk/x/auth/module"
 _ "github.com/cosmos/cosmos-sdk/x/bank/module"
 _ "github.com/cosmos/cosmos-sdk/x/staking/module"
 "github.com/cosmos/cosmos-sdk/core/app"
)

// go:embed app.yaml
var appConfigYAML []byte

func main() {
 app.Run(app.LoadYAML(appConfigYAML))
}

```

## Application to existing SDK modules

So far we have described a system which is largely agnostic to the specifics of the SDK such as store keys, `AppModule`, `BaseApp`, etc. Improvements to these parts of the framework that integrate with the general app wiring framework defined here are described in [ADR 063: Core Module API](#).

## Registration of Inter-Module Hooks

### Registration of Inter-Module Hooks

Some modules define a hooks interface (ex. `StakingHooks`) which allows one module to call back into another module when certain events happen.

With the app wiring framework, these hooks interfaces can be defined as a `OnePerModuleType`s and then the module which consumes these hooks can collect these hooks as a map of module name to hook type (ex. `map[string]FooHooks`). Ex:

```
func init() {
 appmodule.Register(
 &foomodulev1.Module{},
 appmodule.Invoke(InvokeSetFooHooks),
 ...
)
}

func InvokeSetFooHooks(
 keeper *keeper.Keeper,
 fooHooks map[string]FooHooks,
) error {
 for k in sort.Strings(maps.Keys(fooHooks)) {
 keeper.AddFooHooks(fooHooks[k])
 }
}
```

Optionally, the module consuming hooks can allow app's to define an order for calling these hooks based on module name in its config object.

An alternative way for registering hooks via reflection was considered where all keeper types are inspected to see if they implement the hook interface by the modules exposing hooks. This has the downsides of:

- needing to expose all the keepers of all modules to the module providing hooks,
- not allowing for encapsulating hooks on a different type which doesn't expose all keeper methods,
- harder to know statically which module expose hooks or are checking for them.

With the approach proposed here, hooks registration will be obviously observable in `app.go` if `depinject` codegen (described below) is used.

## Code Generation

The `depinject` framework will optionally allow the app configuration and dependency injection wiring to be code generated. This will allow:

- dependency injection wiring to be inspected as regular go code just like the existing `app.go`,
- dependency injection to be opt-in with manual wiring 100% still possible.

Code generation requires that all providers and invokers and their parameters are exported and in non-internal packages.

## Module Semantic Versioning

When we start creating semantically versioned SDK modules that are in standalone go modules, a state machine breaking change to a module should be handled as follows:

- the semantic major version should be incremented, and

- a new semantically versioned module config protobuf type should be created.

For instance, if we have the SDK module for bank in the go module `cosmosdk.io/x/bank` with the module config type `cosmos.bank.module.v1.Module`, and we want to make a state machine breaking change to the module, we would:

- create a new go module `cosmosdk.io/x/bank/v2`,
- with the module config protobuf type `cosmos.bank.module.v2.Module`.

This *does not* mean that we need to increment the protobuf API version for bank. Both modules can support `cosmos.bank.v1`, but `cosmosdk.io/x/bank/v2` will be a separate go module with a separate module config type.

This practice will eventually allow us to use appconfig to load new versions of a module via a configuration change.

Effectively, there should be a 1:1 correspondence between a semantically versioned go module and a versioned module config protobuf type, and major versioning bumps should occur whenever state machine breaking changes are made to a module.

NOTE: SDK modules that are standalone go modules *should not* adopt semantic versioning until the concerns described in [ADR 054: Module Semantic Versioning](#) are addressed. The short-term solution for this issue was left somewhat unresolved. However, the easiest tactic is likely to use a standalone API go module and follow the guidelines described in this comment: <https://github.com/cosmos/cosmos-sdk/pull/11802#issuecomment-1406815181>. For the time-being, it is recommended that Cosmos SDK modules continue to follow tried and true [0-based versioning](#) until an officially recommended solution is provided. This section of the ADR will be updated when that happens and for now, this section should be considered as a design recommendation for future adoption of semantic versioning.

## Consequences

### Backwards Compatibility

Modules which work with the new app wiring system do not need to drop their existing `AppModule` and `NewKeeper` registration paradigms. These two methods can live side-by-side for as long as is needed.

### Positive

- wiring up new apps will be simpler, more succinct and less error-prone
- it will be easier to develop and test standalone SDK modules without needing to replicate all of simapp
- it may be possible to dynamically load modules and upgrade chains without needing to do a coordinated stop and binary upgrade using this mechanism
- easier plugin integration
- dependency injection framework provides more automated reasoning about dependencies in the project, with a graph visualization.

### Negative

- it may be confusing when a dependency is missing although error messages, the GraphViz visualization, and global module registration may help with that

### Neutral

- it will require work and education

## Further Discussions

The protobuf type registration system described in this ADR has not been implemented and may need to be reconsidered in light of code generation. It may be better to do this type registration with a DI provider.

## References

- <https://github.com/cosmos/cosmos-sdk/blob/c3edbb22cab8678c35e21fe0253919996b780c01/simapp/app.go>
- <https://github.com/allinbits/cosmos-sdk-poc>
- <https://github.com/uber-go/dig>
- <https://github.com/google/wire>
- <https://pkg.go.dev/github.com/cosmos/cosmos-sdk/container>
- <https://github.com/cosmos/cosmos-sdk/pull/11802>
- [ADR 063: Core Module API](#)

# ADR 058: Auto-Generated CLI

## Changelog

- 2022-05-04: Initial Draft

## Status

ACCEPTED Partially Implemented

## Abstract

In order to make it easier for developers to write Cosmos SDK modules, we provide infrastructure which automatically generates CLI commands based on protobuf definitions.

## Context

Current Cosmos SDK modules generally implement a CLI command for every transaction and every query supported by the module. These are handwritten for each command and essentially amount to providing some CLI flags or positional arguments for specific fields in protobuf messages.

In order to make sure CLI commands are correctly implemented as well as to make sure that the application works in end-to-end scenarios, we do integration tests using CLI commands. While these tests are valuable on some-level, they can be hard to write and maintain, and run slowly. [Some teams have contemplated](#) moving away from CLI-style integration tests (which are really end-to-end tests) towards narrower integration tests which exercise `MsgClient` and `QueryClient` directly. This might involve replacing the current end-to-end CLI tests with unit tests as there still needs to be some way to test these CLI commands for full quality assurance.

## Decision

To make module development simpler, we provide infrastructure - in the new `client/v2` go module - for automatically generating CLI commands based on protobuf definitions to either replace or complement

handwritten CLI commands. This will mean that when developing a module, it will be possible to skip both writing and testing CLI commands as that can all be taken care of by the framework.

The basic design for automatically generating CLI commands is to:

- create one CLI command for each `rpc` method in a protobuf `Query` or `Msg` service
- create a CLI flag for each field in the `rpc` request type
- for `query` commands call gRPC and print the response as protobuf JSON or YAML (via the `-o` / `--output` flag)
- for `tx` commands, create a transaction and apply common transaction flags

In order to make the auto-generated CLI as easy to use (or easier) than handwritten CLI, we need to do custom handling of specific protobuf field types so that the input format is easy for humans:

- `Coin`, `Coins`, `DecCoin`, and `DecCoins` should be input using the existing format (i.e. `1000uatom`)
- it should be possible to specify an address using either the bech32 address string or a named key in the keyring
- `Timestamp` and `Duration` should accept strings like `2001-01-01T00:00:00Z` and `1h3m` respectively
- pagination should be handled with flags like `--page-limit`, `--page-offset`, etc.
- it should be possible to customize any other protobuf type either via its message name or a `cosmos_proto.scalar` annotation

At a basic level it should be possible to generate a command for a single `rpc` method as well as all the commands for a whole protobuf `service` definition. It should be possible to mix and match auto-generated and handwritten commands.

## Consequences

### Backwards Compatibility

Existing modules can mix and match auto-generated and handwritten CLI commands so it is up to them as to whether they make breaking changes by replacing handwritten commands with slightly different auto-generated ones.

For now the SDK will maintain the existing set of CLI commands for backwards compatibility but new commands will use this functionality.

### Positive

- module developers will not need to write CLI commands
- module developers will not need to test CLI commands
- [lens](#) may benefit from this

### Negative

### Neutral

## Further Discussions

We would like to be able to customize:

- short and long usage strings for commands

- aliases for flags (ex. `-a` for `--amount`)
- which fields are positional parameters rather than flags

It is an [open discussion](#) as to whether these customizations options should live in:

- the .proto files themselves,
- separate config files (ex. YAML), or
- directly in code

Providing the options in .proto files would allow a dynamic client to automatically generate CLI commands on the fly. However, that may pollute the .proto files themselves with information that is only relevant for a small subset of users.

## References

- <https://github.com/regen-network/regen-ledger/issues/1041>
- <https://github.com/cosmos/cosmos-sdk/tree/main/client/v2>
- <https://github.com/cosmos/cosmos-sdk/pull/11725#issuecomment-1108676129>

# ADR 059: Test Scopes

## Changelog

- 2022-08-02: Initial Draft
- 2023-03-02: Add precision for integration tests
- 2023-03-23: Add precision for E2E tests

## Status

PROPOSED Partially Implemented

## Abstract

Recent work in the SDK aimed at breaking apart the monolithic root go module has highlighted shortcomings and inconsistencies in our testing paradigm. This ADR clarifies a common language for talking about test scopes and proposes an ideal state of tests at each scope.

## Context

[ADR-053: Go Module Refactoring](#) expresses our desire for an SDK composed of many independently versioned Go modules, and [ADR-057: App Wiring](#) offers a methodology for breaking apart inter-module dependencies through the use of dependency injection. As described in [EPIC: Separate all SDK modules into standalone go modules](#), module dependencies are particularly complicated in the test phase, where simapp is used as the key test fixture in setting up and running tests. It is clear that the successful completion of Phases 3 and 4 in that EPIC require the resolution of this dependency problem.

In [EPIC: Unit Testing of Modules via Mocks](#) it was thought this Gordian knot could be unwound by mocking all dependencies in the test phase for each module, but seeing how these refactors were complete rewrites of test suites discussions began around the fate of the existing integration tests. One perspective is that they ought to be thrown out, another is that integration tests have some utility of their own and a place in the SDK's testing story.

Another point of confusion has been the current state of CLI test suites, [x/auth](#) for example. In code these are called integration tests, but in reality function as end to end tests by starting up a tendermint node and full application. [EPIC: Rewrite and simplify CLI tests](#) identifies the ideal state of CLI tests using mocks, but does not address the place end to end tests may have in the SDK.

From here we identify three scopes of testing, **unit**, **integration**, **e2e** (end to end), seek to define the boundaries of each, their shortcomings (real and imposed), and their ideal state in the SDK.

## Unit tests

Unit tests exercise the code contained in a single module (e.g. `/x/bank`) or package (e.g. `/client`) in isolation from the rest of the code base. Within this we identify two levels of unit tests, *illustrative* and *journey*. The definitions below lean heavily on [The BDD Books - Formulation](#) section 1.3.

*Illustrative* tests exercise an atomic part of a module in isolation - in this case we might do fixture setup/mocking of other parts of the module.

Tests which exercise a whole module's function with dependencies mocked, are *journeys*. These are almost like integration tests in that they exercise many things together but still use mocks.

Example 1 journey vs illustrative tests - [depinject's BDD style tests](#), show how we can rapidly build up many illustrative cases demonstrating behavioral rules without [very much code](#) while maintaining high level readability.

Example 2 [depinject table driven tests](#)

Example 3 [Bank keeper tests](#) - A mock implementation of `AccountKeeper` is supplied to the keeper constructor.

## Limitations

Certain modules are tightly coupled beyond the test phase. A recent dependency report for `bank -> auth` found 274 total usages of `auth` in `bank`, 50 of which are in production code and 224 in test. This tight coupling may suggest that either the modules should be merged, or refactoring is required to abstract references to the core types tying the modules together. It could also indicate that these modules should be tested together in integration tests beyond mocked unit tests.

In some cases setting up a test case for a module with many mocked dependencies can be quite cumbersome and the resulting test may only show that the mocking framework works as expected rather than working as a functional test of interdependent module behavior.

## Integration tests

Integration tests define and exercise relationships between an arbitrary number of modules and/or application subsystems.

Wiring for integration tests is provided by `depinject` and some [helper code](#) starts up a running application. A section of the running application may then be tested. Certain inputs during different phases of the application life cycle are expected to produce invariant outputs without too much concern for component internals. This type of black box testing has a larger scope than unit testing.

Example 1 [client/grpc\\_query\\_test/TestGRPCQuery](#) - This test is misplaced in `/client`, but tests the life cycle of (at least) `runtime` and `bank` as they progress through startup, genesis and query time. It also exercises the fitness of the client and query server without putting bytes on the wire through the use of [QueryServiceTestHelper](#).

Example 2 `x/evidence` Keeper integration tests - Starts up an application composed of [8 modules](#) with [5 keepers](#) used in the integration test suite. One test in the suite exercises [HandleEquivocationEvidence](#) which contains many interactions with the staking keeper.

Example 3 - Integration suite app configurations may also be specified via golang (not YAML as above) [statically](#) or [dynamically](#).

### Limitations

Setting up a particular input state may be more challenging since the application is starting from a zero state. Some of this may be addressed by good test fixture abstractions with testing of their own. Tests may also be more brittle, and larger refactors could impact application initialization in unexpected ways with harder to understand errors. This could also be seen as a benefit, and indeed the SDK's current integration tests were helpful in tracking down logic errors during earlier stages of app-wiring refactors.

### Simulations

Simulations (also called generative testing) are a special case of integration tests where deterministically random module operations are executed against a running simapp, building blocks on the chain until a specified height is reached. No specific assertions are made for the state transitions resulting from module operations but any error will halt and fail the simulation. Since `crisis` is included in simapp and the simulation runs EndBlockers at the end of each block any module invariant violations will also fail the simulation.

Modules must implement [AppModuleSimulation.WeightedOperations](#) to define their simulation operations. Note that not all modules implement this which may indicate a gap in current simulation test coverage.

Modules not returning simulation operations:

- `auth`
- `evidence`
- `mint`
- `params`

A separate binary, [runsim](#), is responsible for kicking off some of these tests and managing their life cycle.

### Limitations

- [A success](#) may take a long time to run, 7-10 minutes per simulation in CI.
- [Timeouts](#) sometimes occur on apparent successes without any indication why.
- Useful error messages not provided on [failure](#) from CI, requiring a developer to run the simulation locally to reproduce.

### E2E tests

End to end tests exercise the entire system as we understand it in as close an approximation to a production environment as is practical. Presently these tests are located at [tests/e2e](#) and rely on [testutil/network](#) to start up an in-process Tendermint node.

An application should be built as minimally as possible to exercise the desired functionality. The SDK uses an application will only the required modules for the tests. The application developer is advised to use its own application for e2e tests.

### Limitations

In general the limitations of end to end tests are orchestration and compute cost. Scaffolding is required to start up and run a prod-like environment and the this process takes much longer to start and run than unit or integration tests.

Global locks present in Tendermint code cause stateful starting/stopping to sometimes hang or fail intermittently when run in a CI environment.

The scope of e2e tests has been completed with command line interface testing.

## Decision

We accept these test scopes and identify the following decisions points for each.

| Scope       | App Type            | Mocks? |
|-------------|---------------------|--------|
| Unit        | None                | Yes    |
| Integration | integration helpers | Some   |
| Simulation  | minimal app         | No     |
| E2E         | minimal app         | No     |

The decision above is valid for the SDK. An application developer should test their application with their full application instead of the minimal app.

## Unit Tests

All modules must have mocked unit test coverage.

Illustrative tests should outnumber journeys in unit tests.

Unit tests should outnumber integration tests.

Unit tests must not introduce additional dependencies beyond those already present in production code.

When module unit test introduction as per [EPIC: Unit testing of modules via mocks](#) results in a near complete rewrite of an integration test suite the test suite should be retained and moved to `/tests/integration`. We accept the resulting test logic duplication but recommend improving the unit test suite through the addition of illustrative tests.

## Integration Tests

All integration tests shall be located in `/tests/integration`, even those which do not introduce extra module dependencies.

To help limit scope and complexity, it is recommended to use the smallest possible number of modules in application startup, i.e. don't depend on simapp.

Integration tests should outnumber e2e tests.

## Simulations

Simulations shall use a minimal application (usually via app wiring). They are located under `/x/{moduleName}/simulation`.

## E2E Tests

Existing e2e tests shall be migrated to integration tests by removing the dependency on the test network and in-process Tendermint node to ensure we do not lose test coverage.

The e2e rest runner shall transition from in process Tendermint to a runner powered by Docker via [dockertest](#).

E2E tests exercising a full network upgrade shall be written.

The CLI testing aspect of existing e2e tests shall be rewritten using the network mocking demonstrated in [PR#12706](#).

## Consequences

### Positive

- test coverage is increased
- test organization is improved
- reduced dependency graph size in modules
- simapp removed as a dependency from modules
- inter-module dependencies introduced in test code are removed
- reduced CI run time after transitioning away from in process Tendermint

### Negative

- some test logic duplication between unit and integration tests during transition
- test written using dockertest DX may be a bit worse

### Neutral

- some discovery required for e2e transition to dockertest

## Further Discussions

It may be useful if test suites could be run in integration mode (with mocked tendermint) or with e2e fixtures (with real tendermint and many nodes). Integration fixtures could be used for quicker runs, e2e fixtures could be used for more battle hardening.

A PoC `x/gov` was completed in PR [#12847](#) is in progress for unit tests demonstrating BDD [Rejected].

Observing that a strength of BDD specifications is their readability, and a con is the cognitive load while writing and maintaining, current consensus is to reserve BDD use for places in the SDK where complex rules and module interactions are demonstrated. More straightforward or low level test cases will continue to rely on go table tests.

Levels are network mocking in integration and e2e tests are still being worked on and formalized.

# ADR 60: ABCI 1.0 Integration (Phase I)

## Changelog

- 2022-08-10: Initial Draft (@alexanderbez, @tac0turtle)
- Nov 12, 2022: Update `PrepareProposal` and `ProcessProposal` semantics per the initial implementation [PR](#) (@alexanderbez)

## Status

ACCEPTED

## Abstract

This ADR describes the initial adoption of [ABCI 1.0](#), the next evolution of ABCI, within the Cosmos SDK. ABCI 1.0 aims to provide application developers with more flexibility and control over application and consensus semantics, e.g. in-application mempools, in-process oracles, and order-book style matching engines.

## Context

Tendermint will release ABCI 1.0. Notably, at the time of this writing, Tendermint is releasing v0.37.0 which will include `PrepareProposal` and `ProcessProposal`.

The `PrepareProposal` ABCI method is concerned with a block proposer requesting the application to evaluate a series of transactions to be included in the next block, defined as a slice of `TxRecord` objects. The application can either accept, reject, or completely ignore some or all of these transactions. This is an important consideration to make as the application can essentially define and control its own mempool allowing it to define sophisticated transaction priority and filtering mechanisms, by completely ignoring the `TxRecords` Tendermint sends it, favoring its own transactions. This essentially means that the Tendermint mempool acts more like a gossip data structure.

The second ABCI method, `ProcessProposal`, is used to process the block proposer's proposal as defined by `PrepareProposal`. It is important to note the following with respect to `ProcessProposal`:

- Execution of `ProcessProposal` must be deterministic.
- There must be coherence between `PrepareProposal` and `ProcessProposal`. In other words, for any two correct processes  $p$  and  $q$ , if  $q$ 's Tendermint calls `RequestProcessProposal` on  $u_p$ ,  $q$ 's Application returns ACCEPT in `ResponseProcessProposal`.

It is important to note that in ABCI 1.0 integration, the application is NOT responsible for locking semantics - - Tendermint will still be responsible for that. In the future, however, the application will be responsible for locking, which allows for parallel execution possibilities.

## Decision

We will integrate ABCI 1.0, which will be introduced in Tendermint v0.37.0, in the next major release of the Cosmos SDK. We will integrate ABCI 1.0 methods on the `BaseApp` type. We describe the implementations of the two methods individually below.

Prior to describing the implementation of the two new methods, it is important to note that the existing ABCI methods, `CheckTx`, `DeliverTx`, etc, still exist and serve the same functions as they do now.

### `PrepareProposal`

Prior to evaluating the decision for how to implement `PrepareProposal`, it is important to note that `CheckTx` will still be executed and will be responsible for evaluating transaction validity as it does now, with one very important *additive* distinction.

When executing transactions in `CheckTx`, the application will now add valid transactions, i.e. passing the `AnteHandler`, to its own mempool data structure. In order to provide a flexible approach to meet the varying

needs of application developers, we will define both a mempool interface and a data structure utilizing Golang generics, allowing developers to focus only on transaction ordering. Developers requiring absolute full control can implement their own custom mempool implementation.

We define the general mempool interface as follows (subject to change):

```
type Mempool interface {
 // Insert attempts to insert a Tx into the app-side mempool returning
 // an error upon failure.
 Insert(sdk.Context, sdk.Tx) error

 // Select returns an Iterator over the app-side mempool. If txs are specified,
 // then they shall be incorporated into the Iterator. The Iterator must
 // closed by the caller.
 Select(sdk.Context, [][]byte) Iterator

 // CountTx returns the number of transactions currently in the mempool.
 CountTx() int

 // Remove attempts to remove a transaction from the mempool, returning an error
 // upon failure.
 Remove(sdk.Tx) error
}

// Iterator defines an app-side mempool iterator interface that is as minimal as
// possible. The order of iteration is determined by the app-side mempool
// implementation.
type Iterator interface {
 // Next returns the next transaction from the mempool. If there are no more
 // transactions, it returns nil.
 Next() Iterator

 // Tx returns the transaction at the current position of the iterator.
 Tx() sdk.Tx
}
```

We will define an implementation of `Mempool`, defined by `nonceMempool`, that will cover most basic application use-cases. Namely, it will prioritize transactions by transaction sender, allowing for multiple transactions from the same sender.

The default app-side mempool implementation, `nonceMempool`, will operate on a single skip list data structure. Specifically, transactions with the lowest nonce globally are prioritized. Transactions with the same nonce are prioritized by sender address.

```
type nonceMempool struct {
 txQueue *huandu.SkipList
}
```

Previous discussions<sup>1</sup> have come to the agreement that Tendermint will perform a request to the application, via `RequestPrepareProposal`, with a certain amount of transactions reaped from

Tendermint's local mempool. The exact amount of transactions reaped will be determined by a local operator configuration. This is referred to as the "one-shot approach" seen in discussions.

When Tendermint reaps transactions from the local mempool and sends them to the application via `RequestPrepareProposal`, the application will have to evaluate the transactions. Specifically, it will need to inform Tendermint if it should reject and or include each transaction. Note, the application can even replace transactions entirely with other transactions.

When evaluating transactions from `RequestPrepareProposal`, the application will ignore ALL transactions sent to it in the request and instead reap up to `RequestPrepareProposal.max_tx_bytes` from its own mempool.

Since an application can technically insert or inject transactions on `Insert` during `CheckTx` execution, it is recommended that applications ensure transaction validity when reaping transactions during `PrepareProposal`. However, what validity exactly means is entirely determined by the application.

The Cosmos SDK will provide a default `PrepareProposal` implementation that simply select up to `MaxBytes valid` transactions.

However, applications can override this default implementation with their own implementation and set that on `BaseApp` via `SetPrepareProposal`.

### **ProcessProposal**

The `ProcessProposal` ABCI method is relatively straightforward. It is responsible for ensuring validity of the proposed block containing transactions that were selected from the `PrepareProposal` step. However, how an application determines validity of a proposed block depends on the application and its varying use cases. For most applications, simply calling the `AnteHandler` chain would suffice, but there could easily be other applications that need more control over the validation process of the proposed block, such as ensuring txs are in a certain order or that certain transactions are included. While this theoretically could be achieved with a custom `AnteHandler` implementation, it's not the cleanest UX or the most efficient solution.

Instead, we will define an additional ABCI interface method on the existing `Application` interface, similar to the existing ABCI methods such as `BeginBlock` or `EndBlock`. This new interface method will be defined as follows:

```
ProcessProposal(sdk.Context, abci.RequestProcessProposal) error { }
```

Note, we must call `ProcessProposal` with a new internal branched state on the `Context` argument as we cannot simply just use the existing `checkState` because `BaseApp` already has a modified `checkState` at this point. So when executing `ProcessProposal`, we create a similar branched state, `processProposalState`, off of `deliverState`. Note, the `processProposalState` is never committed and is completely discarded after `ProcessProposal` finishes execution.

The Cosmos SDK will provide a default implementation of `ProcessProposal` in which all transactions are validated using the `CheckTx` flow, i.e. the `AnteHandler`, and will always return `ACCEPT` unless any transaction cannot be decoded.

### **DeliverTx**

Since transactions are not truly removed from the app-side mempool during `PrepareProposal`, since `ProcessProposal` can fail or take multiple rounds and we do not want to lose transactions, we need to finally remove the transaction from the app-side mempool during `DeliverTx` since during this phase, the transactions are being included in the proposed block.

Alternatively, we can keep the transactions as truly being removed during the reaping phase in `PrepareProposal` and add them back to the app-side mempool in case `ProcessProposal` fails.

## Consequences

### Backwards Compatibility

ABCI 1.0 is naturally not backwards compatible with prior versions of the Cosmos SDK and Tendermint. For example, an application that requests `RequestPrepareProposal` to the same application that does not speak ABCI 1.0 will naturally fail.

However, in the first phase of the integration, the existing ABCI methods as we know them today will still exist and function as they currently do.

### Positive

- Applications now have full control over transaction ordering and priority.
- Lays the groundwork for the full integration of ABCI 1.0, which will unlock more app-side use cases around block construction and integration with the Tendermint consensus engine.

### Negative

- Requires that the "mempool", as a general data structure that collects and stores uncommitted transactions will be duplicated between both Tendermint and the Cosmos SDK.
- Additional requests between Tendermint and the Cosmos SDK in the context of block execution. Albeit, the overhead should be negligible.
- Not backwards compatible with previous versions of Tendermint and the Cosmos SDK.

## Further Discussions

It is possible to design the app-side implementation of the `Mempool[T MempoolTx]` in many different ways using different data structures and implementations. All of which have different tradeoffs. The proposed solution keeps things simple and covers cases that would be required for most basic applications. There are tradeoffs that can be made to improve performance of reaping and inserting into the provided mempool implementation.

## References

- <https://github.com/tendermint/tendermint/blob/master/spec/abci%2B%2B/README.md>
- [1] <https://github.com/tendermint/tendermint/issues/7750#issuecomment-1076806155>
- [2] <https://github.com/tendermint/tendermint/issues/7750#issuecomment-1075717151>

# ADR ADR-061: Liquid Staking

## Changelog

- 2022-09-10: Initial Draft (@zmanian)

## Status

ACCEPTED

## Abstract

Add a semi-fungible liquid staking primitive to the default Cosmos SDK staking module. This upgrades proof of stake to enable safe designs with lower overall monetary issuance and integration with numerous liquid staking protocols like Stride, Persistence, Quicksilver, Lido etc.

## Context

The original release of the Cosmos Hub featured the implementation of a ground breaking proof of stake mechanism featuring delegation, slashing, in protocol reward distribution and adaptive issuance. This design was state of the art for 2016 and has been deployed without major changes by many L1 blockchains.

As both Proof of Stake and blockchain use cases have matured, this design has aged poorly and should no longer be considered a good baseline Proof of Stake issuance. In the world of application specific blockchains, there cannot be a one size fits all blockchain but the Cosmos SDK does endeavour to provide a good baseline implementation and one that is suitable for the Cosmos Hub.

The most important deficiency of the legacy staking design is that it composes poorly with on chain protocols for trading, lending, derivatives that are referred to collectively as DeFi. The legacy staking implementation starves these applications of liquidity by increasing the risk free rate adaptively. It basically makes DeFi and staking security somewhat incompatible.

The Osmosis team has adopted the idea of Superfluid and Interfluid staking where assets that are participating in DeFi applications can also be used in proof of stake. This requires tight integration with an enshrined set of DeFi applications and thus is unsuitable for the Cosmos SDK.

It's also important to note that Interchain Accounts are available in the default IBC implementation and can be used to [rehypothecate](#) delegations. Thus liquid staking is already possible and these changes merely improve the UX of liquid staking. Centralized exchanges also rehypothecate staked assets, posing challenges for decentralization. This ADR takes the position that adoption of in-protocol liquid staking is the preferable outcome and provides new levers to incentivize decentralization of stake.

These changes to the staking module have been in development for more than a year and have seen substantial industry adoption who plan to build staking UX. The internal economics at Informal team has also done a review of the impacts of these changes and this review led to the development of the exempt delegation system. This system provides governance with a tuneable parameter for modulating the risks of principal agent problem called the exemption factor.

## Decision

We implement the semi-fungible liquid staking system and exemption factor system within the cosmos sdk. Though registered as fungible assets, these tokenized shares have extremely limited fungibility, only among the specific delegation record that was created when shares were tokenized. These assets can be used for OTC trades but composability with DeFi is limited. The primary expected use case is improving the user experience of liquid staking providers.

A new governance parameter is introduced that defines the ratio of exempt to issued tokenized shares. This is called the exemption factor. A larger exemption factor allows more tokenized shares to be issued for a

smaller amount of exempt delegations. If governance is comfortable with how the liquid staking market is evolving, it makes sense to increase this value.

Min self delegation is removed from the staking system with the expectation that it will be replaced by the exempt delegations system. The exempt delegation system allows multiple accounts to demonstrate economic alignment with the validator operator as team members, partners etc. without co-mingling funds. Delegation exemption will likely be required to grow the validators' business under widespread adoption of liquid staking once governance has adjusted the exemption factor.

When shares are tokenized, the underlying shares are transferred to a module account and rewards go to the module account for the TokenizedShareRecord.

There is no longer a mechanism to override the validators vote for TokenizedShares.

#### **MsgTokenizeShares**

The MsgTokenizeShares message is used to create tokenize delegated tokens. This message can be executed by any delegator who has positive amount of delegation and after execution the specific amount of delegation disappear from the account and share tokens are provided. Share tokens are denominated in the validator and record id of the underlying delegation.

A user may tokenize some or all of their delegation.

They will receive shares with the denom of `cosmosvaloper1xxxxx/5` where 5 is the record id for the validator operator.

MsgTokenizeShares fails if the account is a VestingAccount. Users will have to move vested tokens to a new account and endure the unbonding period. We view this as an acceptable tradeoff vs. the complex book keeping required to track vested tokens.

The total amount of outstanding tokenized shares for the validator is checked against the sum of exempt delegations multiplied by the exemption factor. If the tokenized shares exceeds this limit, execution fails.

MsgTokenizeSharesResponse provides the number of tokens generated and their denom.

#### **MsgRedeemTokensforShares**

The MsgRedeemTokensforShares message is used to redeem the delegation from share tokens. This message can be executed by any user who owns share tokens. After execution delegations will appear to the user.

#### **MsgTransferTokenizeShareRecord**

The MsgTransferTokenizeShareRecord message is used to transfer the ownership of rewards generated from the tokenized amount of delegation. The tokenize share record is created when a user tokenize his/her delegation and deleted when the full amount of share tokens are redeemed.

This is designed to work with liquid staking designs that do not redeem the tokenized shares and may instead want to keep the shares tokenized.

#### **MsgExemptDelegation**

The MsgExemptDelegation message is used to exempt a delegation to a validator. If the exemption factor is greater than 0, this will allow more delegation shares to be issued from the validator.

This design allows the chain to force an amount of self-delegation by validators participating in liquid staking schemes.

## Consequences

### Backwards Compatibility

By setting the exemption factor to zero, this module works like legacy staking. The only substantial change is the removal of min-self-bond and without any tokenized shares, there is no incentive to exempt delegation.

### Positive

This approach should enable integration with liquid staking providers and improved user experience. It provides a pathway to security under non-exponential issuance policies in the baseline staking module.

## ADR 062: Collections, a simplified storage layer for cosmos-sdk modules.

### Changelog

- 30/11/2022: PROPOSED

### Status

PROPOSED - Implemented

### Abstract

We propose a simplified module storage layer which leverages golang generics to allow module developers to handle module storage in a simple and straightforward manner, whilst offering safety, extensibility and standardisation.

### Context

Module developers are forced into manually implementing storage functionalities in their modules, those functionalities include but are not limited to:

- Defining key to bytes formats.
- Defining value to bytes formats.
- Defining secondary indexes.
- Defining query methods to expose outside to deal with storage.
- Defining local methods to deal with storage writing.
- Dealing with genesis imports and exports.
- Writing tests for all the above.

This brings in a lot of problems:

- It blocks developers from focusing on the most important part: writing business logic.
- Key to bytes formats are complex and their definition is error-prone, for example:
  - how do I format time to bytes in such a way that bytes are sorted?

- o how do I ensure when I don't have namespace collisions when dealing with secondary indexes?
- The lack of standardisation makes life hard for clients, and the problem is exacerbated when it comes to providing proofs for objects present in state. Clients are forced to maintain a list of object paths to gather proofs.

## Current Solution: ORM

The current SDK proposed solution to this problem is [ORM](#). Whilst ORM offers a lot of good functionality aimed at solving these specific problems, it has some downsides:

- It requires migrations.
- It uses the newest protobuf golang API, whilst the SDK still mainly uses gogoproto.
- Integrating ORM into a module would require the developer to deal with two different golang frameworks (golang protobuf + gogoproto) representing the same API objects.
- It has a high learning curve, even for simple storage layers as it requires developers to have knowledge around protobuf options, custom cosmos-sdk storage extensions, and tooling download. Then after this they still need to learn the code-generated API.

## CosmWasm Solution: cw-storage-plus

The collections API takes inspiration from [cw-storage-plus](#), which has demonstrated to be a powerful tool for dealing with storage in CosmWasm contracts. It's simple, does not require extra tooling, it makes it easy to deal with complex storage structures (indexes, snapshot, etc). The API is straightforward and explicit.

## Decision

We propose to port the `collections` API, whose implementation lives in [NibiruChain/collections](#) to cosmos-sdk.

Collections implements four different storage handlers types:

- `Map` : which deals with simple `key=>object` mappings.
- `KeySet` : which acts as a `Set` and only retains keys and no object (usecase: allow-lists).
- `Item` : which always contains only one object (usecase: Params)
- `Sequence` : which implements a simple always increasing number (usecase: Nonces)
- `IndexedMap` : builds on top of `Map` and `KeySet` and allows to create relationships with `Objects` and `Objects` secondary keys.

All the collection APIs build on top of the simple `Map` type.

Collections is fully generic, meaning that anything can be used as `Key` and `Value`. It can be a protobuf object or not.

Collections types, in fact, delegate the duty of serialisation of keys and values to a secondary collections API component called `ValueEncoders` and `KeyEncoders`.

`ValueEncoders` take care of converting a value to bytes (relevant only for `Map`). And offers a plug and play layer which allows us to change how we encode objects, which is relevant for swapping serialisation frameworks and enhancing performance. Collections already comes in with default `ValueEncoders`, specifically for: protobuf objects, special SDK types (`sdk.Int`, `sdk.Dec`).

`KeyEncoders` take care of converting keys to bytes, `collections` already comes in with some default `KeyEncoders` for some primitive golang types (`uint64`, `string`, `time.Time`, ...) and some widely used sdk

types (sdk.Acc/Val/ConsAddress, sdk.Int/Dec, ...). These default implementations also offer safety around proper lexicographic ordering and namespace-collision.

Examples of the collections API can be found here:

- introduction: <https://github.com/NibiruChain/collections/tree/main/examples>
- usage in nibiru: [x/oracle](#), [x/perp](#)
- cosmos-sdk's x/staking migrated: <https://github.com/testinginprod/cosmos-sdk/pull/22>

## Consequences

### Backwards Compatibility

The design of `ValueEncoders` and `KeyEncoders` allows modules to retain the same `byte(key) => byte(value)` mappings, making the upgrade to the new storage layer non-state breaking.

### Positive

- ADR aimed at removing code from the SDK rather than adding it. Migrating just `x/staking` to collections would yield to a net decrease in LOC (even considering the addition of collections itself).
- Simplifies and standardises storage layers across modules in the SDK.
- Does not require to have to deal with protobuf.
- It's pure golang code.
- Generalisation over `KeyEncoders` and `ValueEncoders` allows us to not tie ourself to the data serialisation framework.
- `KeyEncoders` and `ValueEncoders` can be extended to provide schema reflection.

### Negative

- Golang generics are not as battle-tested as other Golang features, despite being used in production right now.
- Collection types instantiation needs to be improved.

### Neutral

{neutral consequences}

## Further Discussions

- Automatic genesis import/export (not implemented because of API breakage)
- Schema reflection

## References

## ADR 063: Core Module API

### Changelog

- 2022-08-18 First Draft
- 2022-12-08 First Draft
- 2023-01-24 Updates

## Status

ACCEPTED Partially Implemented

## Abstract

A new core API is proposed as a way to develop cosmos-sdk applications that will eventually replace the existing `AppModule` and `sdk.Context` frameworks a set of core services and extension interfaces. This core API aims to:

- be simpler
- more extensible
- more stable than the current framework
- enable deterministic events and queries,
- support event listeners
- [ADR 033: Protobuf-based Inter-Module Communication](#) clients.

## Context

Historically modules have exposed their functionality to the framework via the `AppModule` and `AppModuleBasic` interfaces which have the following shortcomings:

- both `AppModule` and `AppModuleBasic` need to be defined and registered which is counter-intuitive
- apps need to implement the full interfaces, even parts they don't need (although there are workarounds for this),
- interface methods depend heavily on unstable third party dependencies, in particular Comet,
- legacy required methods have littered these interfaces for far too long

In order to interact with the state machine, modules have needed to do a combination of these things:

- get store keys from the app
- call methods on `sdk.Context` which contains more or less the full set of capability available to modules.

By isolating all the state machine functionality into `sdk.Context`, the set of functionalities available to modules are tightly coupled to this type. If there are changes to upstream dependencies (such as Comet) or new functionalities are desired (such as alternate store types), the changes need impact `sdk.Context` and all consumers of it (basically all modules). Also, all modules now receive `context.Context` and need to convert these to `sdk.Context`'s with a non-ergonomic unwrapping function.

Any breaking changes to these interfaces, such as ones imposed by third-party dependencies like Comet, have the side effect of forcing all modules in the ecosystem to update in lock-step. This means it is almost impossible to have a version of the module which can be run with 2 or 3 different versions of the SDK or 2 or 3 different versions of another module. This lock-step coupling slows down overall development within the ecosystem and causes updates to components to be delayed longer than they would if things were more stable and loosely coupled.

## Decision

The `core` API proposes a set of core APIs that modules can rely on to interact with the state machine and expose their functionalities to it that are designed in a principled way such that:

- tight coupling of dependencies and unrelated functionalities is minimized or eliminated
- APIs can have long-term stability guarantees
- the SDK framework is extensible in a safe and straightforward way

The design principles of the core API are as follows:

- everything that a module wants to interact with in the state machine is a service
- all services coordinate state via `context.Context` and don't try to recreate the "bag of variables" approach of `sdk.Context`
- all independent services are isolated in independent packages with minimal APIs and minimal dependencies
- the core API should be minimalistic and designed for long-term support (LTS)
- a "runtime" module will implement all the "core services" defined by the core API and can handle all module functionalities exposed by core extension interfaces
- other non-core and/or non-LTS services can be exposed by specific versions of runtime modules or other modules following the same design principles, this includes functionality that interacts with specific non-stable versions of third party dependencies such as Comet
- the core API doesn't implement *any* functionality, it just defines types
- go stable API compatibility guidelines are followed: <https://go.dev/blog/module-compatibility>

A "runtime" module is any module which implements the core functionality of composing an ABCI app, which is currently handled by `BaseApp` and the `ModuleManager`. Runtime modules which implement the core API are *intentionally* separate from the core API in order to enable more parallel versions and forks of the runtime module than is possible with the SDK's current tightly coupled `BaseApp` design while still allowing for a high degree of composability and compatibility.

Modules which are built only against the core API don't need to know anything about which version of runtime, `BaseApp` or Comet in order to be compatible. Modules from the core mainline SDK could be easily composed with a forked version of runtime with this pattern.

This design is intended to enable matrices of compatible dependency versions. Ideally a given version of any module is compatible with multiple versions of the runtime module and other compatible modules. This will allow dependencies to be selectively updated based on battle-testing. More conservative projects may want to update some dependencies slower than more fast moving projects.

## Core Services

The following "core services" are defined by the core API. All valid runtime module implementations should provide implementations of these services to modules via both [dependency injection](#) and manual wiring. The individual services described below are all bundled in a convenient `appmodule.Service` "bundle service" so that for simplicity modules can declare a dependency on a single service.

### Store Services

Store services will be defined in the `cosmosdk.io/core/store` package.

The generic `store.KVStore` interface is the same as current SDK `KVStore` interface. Store keys have been refactored into store services which, instead of expecting the context to know about stores, invert the pattern and allow retrieving a store from a generic context. There are three store services for the three types of currently supported stores - regular kv-store, memory, and transient:

```
type KVStoreService interface {
 OpenKVStore(context.Context) KVStore
```

```

}

type MemoryStoreService interface {
 OpenMemoryStore(context.Context) KVStore
}

type TransientStoreService interface {
 OpenTransientStore(context.Context) KVStore
}

```

Modules can use these services like this:

```

func (k msgServer) Send(ctx context.Context, msg *types.MsgSend)
(*types.MsgSendResponse, error) {
 store := k.kvStoreSvc.OpenKVStore(ctx)
}

```

Just as with the current runtime module implementation, modules will not need to explicitly name these store keys, but rather the runtime module will choose an appropriate name for them and modules just need to request the type of store they need in their dependency injection (or manual) constructors.

## Event Service

The event `Service` will be defined in the `cosmosdk.io/core/event` package.

The event `Service` allows modules to emit typed and legacy untyped events:

```

package event

type Service interface {
 // EmitProtoEvent emits events represented as a protobuf message (as described in
 // ADR 032).
 //
 // Callers SHOULD assume that these events may be included in consensus. These
 // events
 // MUST be emitted deterministically and adding, removing or changing these events
 // SHOULD
 // be considered state-machine breaking.
 EmitProtoEvent(ctx context.Context, event protoiface.MessageV1) error

 // EmitKEvent emits an event based on an event and kv-pair attributes.
 //
 // These events will not be part of consensus and adding, removing or changing
 // these events is
 // not a state-machine breaking change.
 EmitKEvent(ctx context.Context, eventType string, attrs ...KEventAttribute)
 error

 // EmitProtoEventNonConsensus emits events represented as a protobuf message (as
 // described in ADR 032), without
 // including it in blockchain consensus.
 //
}

```

```

 // These events will not be part of consensus and adding, removing or changing
events is
 // not a state-machine breaking change.
 EmitProtoEventNonConsensus(ctx context.Context, event protoiface.MessageV1) error
}

```

Typed events emitted with `EmitProto` should be assumed to be part of blockchain consensus (whether they are part of the block or app hash is left to the runtime to specify).

Events emitted by `EmitKVEvent` and `EmitProtoEventNonConsensus` are not considered to be part of consensus and cannot be observed by other modules. If there is a client-side need to add events in patch releases, these methods can be used.

## Logger

A logger (`cosmosdk.io/log`) must be supplied using `depinject`, and will be made available for modules to use via `depinject.In`. Modules using it should follow the current pattern in the SDK by adding the module name before using it.

```

type ModuleInputs struct {
 depinjection.In

 Logger log.Logger
}

func ProvideModule(in ModuleInputs) ModuleOutputs {
 keeper := keeper.NewKeeper(
 in.logger,
)
}

func NewKeeper(logger log.Logger) Keeper {
 return Keeper{
 logger: logger.With(log.ModuleKey, "x/" + types.ModuleName),
 }
}

```

```
Core ` AppModule` extension interfaces
```

Modules will provide their core services to the runtime module via extension interfaces built on top of the ``cosmosdk.io/core/appmodule.AppModule`` tag interface. This tag interface requires only two empty methods which allow `depinject` to identify implementors as `depinject.OnePerModule` types and as app module implementations:

```

```go
type AppModule interface {
    depinjection.OnePerModuleType
}

```

```
// Is AppModule is a dummy method to tag a struct as implementing an AppModule.  
Is AppModule()  
}
```

Other core extension interfaces will be defined in `cosmosdk.io/core` should be supported by valid runtime implementations.

MsgServer and QueryServer registration

`MsgServer` and `QueryServer` registration is done by implementing the `HasServices` extension interface:

```
type HasServices interface {  
    AppModule  
  
    RegisterServices(grpc.ServiceRegistrar)  
}
```

Because of the `cosmos.msg.v1.service` protobuf option, required for `Msg` services, the same `ServiceRegistrar` can be used to register both `Msg` and query services.

Genesis

The genesis `Handler` functions - `DefaultGenesis`, `ValidateGenesis`, `InitGenesis` and `ExportGenesis` - are specified against the `GenesisSource` and `GenesisTarget` interfaces which will abstract over genesis sources which may be a single JSON object or collections of JSON objects that can be efficiently streamed.

```
// GenesisSource is a source for genesis data in JSON format. It may abstract over a  
// single JSON object or separate files for each field in a JSON object that can  
// be streamed over. Modules should open a separate io.ReadCloser for each field  
that  
// is required. When fields represent arrays they can efficiently be streamed  
// over. If there is no data for a field, this function should return nil, nil. It  
is  
// important that the caller closes the reader when done with it.  
type GenesisSource = func(field string) (io.ReadCloser, error)  
  
// GenesisTarget is a target for writing genesis data in JSON format. It may  
// abstract over a single JSON object or JSON in separate files that can be  
// streamed over. Modules should open a separate io.WriteCloser for each field  
// and should prefer writing fields as arrays when possible to support efficient  
// iteration. It is important the caller closes the writer AND checks the error  
// when done with it. It is expected that a stream of JSON data is written  
// to the writer.  
type GenesisTarget = func(field string) (io.WriteCloser, error)
```

All genesis objects for a given module are expected to conform to the semantics of a JSON object. Each field in the JSON object should be read and written separately to support streaming genesis. The [ORM](#) and

[collections](#) both support streaming genesis and modules using these frameworks generally do not need to write any manual genesis code.

To support genesis, modules should implement the `HasGenesis` extension interface:

```
type HasGenesis interface {
    AppModule

    // DefaultGenesis writes the default genesis for this module to the target.
    DefaultGenesis(GenesisTarget) error

    // ValidateGenesis validates the genesis data read from the source.
    ValidateGenesis(GenesisSource) error

    // InitGenesis initializes module state from the genesis source.
    InitGenesis(context.Context, GenesisSource) error

    // ExportGenesis exports module state to the genesis target.
    ExportGenesis(context.Context, GenesisTarget) error
}
```

Begin and End Blockers

Modules that have functionality that runs before transactions (begin blockers) or after transactions (end blockers) should implement the has `HasBeginBlocker` and/or `HasEndBlocker` interfaces:

```
type HasBeginBlocker interface {
    AppModule
    BeginBlock(context.Context) error
}

type HasEndBlocker interface {
    AppModule
    EndBlock(context.Context) error
}
```

The `BeginBlock` and `EndBlock` methods will take a `context.Context`, because:

- most modules don't need Comet information other than `BlockInfo` so we can eliminate dependencies on specific Comet versions
- for the few modules that need Comet block headers and/or return validator updates, specific versions of the runtime module will provide specific functionality for interacting with the specific version(s) of Comet supported

In order for `BeginBlock`, `EndBlock` and `InitGenesis` to send back validator updates and retrieve full Comet block headers, the runtime module for a specific version of Comet could provide services like this:

```
type ValidatorUpdateService interface {
    SetValidatorUpdates(context.Context, []abci.ValidatorUpdate)
}
```

Header Service defines a way to get header information about a block. This information is generalized for all implementations:

```
type Service interface {
    GetHeaderInfo(context.Context) Info
}

type Info struct {
    Height int64           // Height returns the height of the block
    Hash []byte             // Hash returns the hash of the block header
    Time time.Time          // Time returns the time of the block
    ChainID string          // ChainID returns the chain ID of the block
}
```

Comet Service provides a way to get comet specific information:

```
type Service interface {
    GetCometInfo(context.Context) Info
}

type CometInfo struct {
    Evidence []abci.Misbehavior // Misbehavior returns the misbehavior of the block
    // ValidatorsHash returns the hash of the validators
    // For Comet, it is the hash of the next validators
    ValidatorsHash []byte
    ProposerAddress []byte      // ProposerAddress returns the address of the
                                // block proposer
    DecidedLastCommit abci.CommitInfo // DecidedLastCommit returns the last commit
info
}
```

If a user would like to provide a module other information they would need to implement another service like:

```
type RollKit Interface {
    ...
}
```

We know these types will change at the Comet level and that also a very limited set of modules actually need this functionality, so they are intentionally kept out of core to keep core limited to the necessary, minimal set of stable APIs.

Remaining Parts of AppModule

The current `AppModule` framework handles a number of additional concerns which aren't addressed by this core API. These include:

- gas
- block headers
- upgrades

- registration of gogo proto and amino interface types
- cobra query and tx commands
- gRPC gateway
- crisis module invariants
- simulations

Additional `AppModule` extension interfaces either inside or outside of core will need to be specified to handle these concerns.

In the case of gogo proto and amino interfaces, the registration of these generally should happen as early as possible during initialization and in [ADR 057: App Wiring](#), protobuf type registration happens before dependency injection (although this could alternatively be done dedicated DI providers).

gRPC gateway registration should probably be handled by the runtime module, but the core API shouldn't depend on gRPC gateway types as 1) we are already using an older version and 2) it's possible the framework can do this registration automatically in the future. So for now, the runtime module should probably provide some sort of specific type for doing this registration ex:

```
type GrpcGatewayInfo struct {
    Handlers []GrpcGatewayHandler
}

type GrpcGatewayHandler func(ctx context.Context, mux *runtime.ServeMux, client
QueryClient) error
```

which modules can return in a provider:

```
func ProvideGrpcGateway() GrpcGatewayInfo {
    return GrpcGatewayinfo {
        Handlers: []Handler {types.RegisterQueryHandlerClient}
    }
}
```

Crisis module invariants and simulations are subject to potential redesign and should be managed with types defined in the crisis and simulation modules respectively.

Extension interface for CLI commands will be provided via the `cosmosdk.io/client/v2` module and its [autocli](#) framework.

Example Usage

Here is an example of setting up a hypothetical `foo v2` module which uses the [ORM](#) for its state management and genesis.

```
type Keeper struct {
    db orm.ModuleDB
    evtSrv event.Service
}

func (k Keeper) RegisterServices(r grpc.ServiceRegistrar) {
    foov1.RegisterMsgServer(r, k)
```

```

    foov1.RegisterQueryServer(r, k)
}

func (k Keeper) BeginBlock(context.Context) error {
    return nil
}

func ProvideApp(config *foomodulev2.Module, evtSvc event.EventService, db
orm.ModuleDB) (Keeper, appmodule.AppModule){
    k := &Keeper{db: db, evtSvc: evtSvc}
    return k, k
}

```

Runtime Compatibility Version

The `core` module will define a static integer var, `cosmosdk.io/core.RuntimeCompatibilityVersion`, which is a minor version indicator of the core module that is accessible at runtime. Correct runtime module implementations should check this compatibility version and return an error if the current `RuntimeCompatibilityVersion` is higher than the version of the core API that this runtime version can support. When new features are added to the `core` module API that runtime modules are required to support, this version should be incremented.

Runtime Modules

The initial `runtime` module will simply be created within the existing `github.com/cosmos/cosmos-sdk` go module under the `runtime` package. This module will be a small wrapper around the existing `BaseApp`, `sdk.Context` and module manager and follow the Cosmos SDK's existing [0-based versioning](#). To move to semantic versioning as well as runtime modularity, new officially supported runtime modules will be created under the `cosmosdk.io/runtime` prefix. For each supported consensus engine a semantically-versioned go module should be created with a runtime implementation for that consensus engine. For example:

- `cosmosdk.io/runtime/comet`
- `cosmosdk.io/runtime/comet/v2`
- `cosmosdk.io/runtime/rollkit`
- etc.

These runtime modules should attempt to be semantically versioned even if the underlying consensus engine is not. Also, because a runtime module is also a first class Cosmos SDK module, it should have a protobuf module config type. A new semantically versioned module config type should be created for each of these runtime module such that there is a 1:1 correspondence between the go module and module config type. This is the same practice should be followed for every semantically versioned Cosmos SDK module as described in [ADR 057: App Wiring](#).

Currently, `github.com/cosmos/cosmos-sdk/runtime` uses the protobuf config type `cosmos.app.runtime.v1alpha1.Module`. When we have a standalone v1 comet runtime, we should use a dedicated protobuf module config type such as `cosmos.runtime.comet.v1.Module1`. When we release v2 of the comet runtime (`cosmosdk.io/runtime/comet/v2`) we should have a corresponding `cosmos.runtime.comet.v2.Module` protobuf type.

In order to make it easier to support different consensus engines that support the same core module functionality as described in this ADR, a common go module should be created with shared runtime

components. The easiest runtime components to share initially are probably the message/query router, inter-module client, service register, and event router. This common runtime module should be created initially as the `cosmosdk.io/runtime/common` go module.

When this new architecture has been implemented, the main dependency for a Cosmos SDK module would be `cosmosdk.io/core` and that module should be able to be used with any supported consensus engine (to the extent that it does not explicitly depend on consensus engine specific functionality such as Comet's block headers). An app developer would then be able to choose which consensus engine they want to use by importing the corresponding runtime module. The current `BaseApp` would be refactored into the `cosmosdk.io/runtime/comet` module, the router infrastructure in `baseapp/` would be refactored into `cosmosdk.io/runtime/common` and support ADR 033, and eventually a dependency on `github.com/cosmos/cosmos-sdk` would no longer be required.

In short, modules would depend primarily on `cosmosdk.io/core`, and each `cosmosdk.io/runtime/{consensus-engine}` would implement the `cosmosdk.io/core` functionality for that consensus engine.

One additional piece that would need to be resolved as part of this architecture is how runtimes relate to the server. Likely it would make sense to modularize the current server architecture so that it can be used with any runtime even if that is based on a consensus engine besides Comet. This means that eventually the Comet runtime would need to encapsulate the logic for starting Comet and the ABCI app.

Testing

A mock implementation of all services should be provided in core to allow for unit testing of modules without needing to depend on any particular version of runtime. Mock services should allow tests to observe service behavior or provide a non-production implementation - for instance memory stores can be used to mock stores.

For integration testing, a mock runtime implementation should be provided that allows composing different app modules together for testing without a dependency on runtime or Comet.

Consequences

Backwards Compatibility

Early versions of runtime modules should aim to support as much as possible modules built with the existing `AppModule` / `sdk.Context` framework. As the core API is more widely adopted, later runtime versions may choose to drop support and only support the core API plus any runtime module specific APIs (like specific versions of Comet).

The core module itself should strive to remain at the go semantic version `v1` as long as possible and follow design principles that allow for strong long-term support (LTS).

Older versions of the SDK can support modules built against core with adaptors that convert wrap core `AppModule` implementations in implementations of `AppModule` that conform to that version of the SDK's semantics as well as by providing service implementations by wrapping `sdk.Context`.

Positive

- better API encapsulation and separation of concerns
- more stable APIs
- more framework extensibility

- deterministic events and queries
- event listeners
- inter-module msg and query execution support
- more explicit support for forking and merging of module versions (including runtime)

Negative

Neutral

- modules will need to be refactored to use this API
- some replacements for `AppModule` functionality still need to be defined in follow-ups (type registration, commands, invariants, simulations) and this will take additional design work

Further Discussions

- gas
- block headers
- upgrades
- registration of gogo proto and amino interface types
- cobra query and tx commands
- gRPC gateway
- crisis module invariants
- simulations

References

- [ADR 033: Protobuf-based Inter-Module Communication](#)
- [ADR 057: App Wiring](#)
- [ADR 055: ORM](#)
- [ADR 028: Public Key Addresses](#)
- [Keeping Your Modules Compatible](#)

ADR 64: ABCI 2.0 Integration (Phase II)

Changelog

- 2023-01-17: Initial Draft (@alexanderbez)
- 2023-04-06: Add upgrading section (@alexanderbez)
- 2023-04-10: Simplify vote extension state persistence (@alexanderbez)
- 2023-07-07: Revise vote extension state persistence (@alexanderbez)

Status

ACCEPTED

Abstract

This ADR outlines the continuation of the efforts to implement ABCI++ in the Cosmos SDK outlined in [ADR 060: ABCI 1.0 \(Phase I\)](#).

Specifically, this ADR outlines the design and implementation of ABCI 2.0, which includes `ExtendVote`, `VerifyVoteExtension` and `FinalizeBlock`.

Context

ABCI 2.0 continues the promised updates from ABCI++, specifically three additional ABCI methods that the application can implement in order to gain further control, insight and customization of the consensus process, unlocking many novel use-cases that previously not possible. We describe these three new methods below:

ExtendVote

This method allows each validator process to extend the pre-commit phase of the CometBFT consensus process. Specifically, it allows the application to perform custom business logic that extends the pre-commit vote and supply additional data as part of the vote, although they are signed separately by the same key.

The data, called vote extension, will be broadcast and received together with the vote it is extending, and will be made available to the application in the next height. Specifically, the proposer of the next block will receive the vote extensions in `RequestPrepareProposal.local_last_commit.votes`.

If the application does not have vote extension information to provide, it returns a 0-length byte array as its vote extension.

NOTE:

- Although each validator process submits its own vote extension, ONLY the *proposer* of the *next* block will receive all the vote extensions included as part of the pre-commit phase of the previous block. This means only the proposer will implicitly have access to all the vote extensions, via `RequestPrepareProposal`, and that not all vote extensions may be included, since a validator does not have to wait for all pre-commits, only 2/3.
- The pre-commit vote is signed independently from the vote extension.

VerifyVoteExtension

This method allows validators to validate the vote extension data attached to each pre-commit message it receives. If the validation fails, the whole pre-commit message will be deemed invalid and ignored by CometBFT.

CometBFT uses `VerifyVoteExtension` when validating a pre-commit vote. Specifically, for a pre-commit, CometBFT will:

- Reject the message if it doesn't contain a signed vote AND a signed vote extension
- Reject the message if the vote's signature OR the vote extension's signature fails to verify
- Reject the message if `VerifyVoteExtension` was rejected by the app

Otherwise, CometBFT will accept the pre-commit message.

Note, this has important consequences on liveness, i.e., if vote extensions repeatedly cannot be verified by correct validators, CometBFT may not be able to finalize a block even if sufficiently many (+2/3) validators send pre-commit votes for that block. Thus, `VerifyVoteExtension` should be used with special care.

CometBFT recommends that an application that detects an invalid vote extension SHOULD accept it in `ResponseVerifyVoteExtension` and ignore it in its own logic.

FinalizeBlock

This method delivers a decided block to the application. The application must execute the transactions in the block deterministically and update its state accordingly. Cryptographic commitments to the block and transaction results, returned via the corresponding parameters in `ResponseFinalizeBlock`, are included in the header of the next block. CometBFT calls it when a new block is decided.

In other words, `FinalizeBlock` encapsulates the current ABCI execution flow of `BeginBlock`, one or more `DeliverTx`, and `EndBlock` into a single ABCI method. CometBFT will no longer execute requests for these legacy methods and instead will just simply call `FinalizeBlock`.

Decision

We will discuss changes to the Cosmos SDK to implement ABCI 2.0 in two distinct phases, `VoteExtensions` and `FinalizeBlock`.

`VoteExtensions`

Similarly for `PrepareProposal` and `ProcessProposal`, we propose to introduce two new handlers that an application can implement in order to provide and verify vote extensions.

We propose the following new handlers for applications to implement:

```
type ExtendVoteHandler func(sdk.Context, abci.RequestExtendVote)
abci.ResponseExtendVote
type VerifyVoteExtensionHandler func(sdk.Context, abci.RequestVerifyVoteExtension)
abci.ResponseVerifyVoteExtension
```

A new execution state, `voteExtensionState`, will be introduced and provided as the `Context` that is supplied to both handlers. It will contain relevant metadata such as the block height and block hash. Note, `voteExtensionState` is never committed and will exist as ephemeral state only in the context of a single block.

If an application decides to implement `ExtendVoteHandler`, it must return a non-nil `ResponseExtendVote.VoteExtension`.

Recall, an implementation of `ExtendVoteHandler` does NOT need to be deterministic, however, given a set of vote extensions, `VerifyVoteExtensionHandler` must be deterministic, otherwise the chain may suffer from liveness faults. In addition, recall CometBFT proceeds in rounds for each height, so if a decision cannot be made about a block proposal at a given height, CometBFT will proceed to the next round and thus will execute `ExtendVote` and `VerifyVoteExtension` again for the new round for each validator until 2/3 valid pre-commits can be obtained.

Given the broad scope of potential implementations and use-cases of vote extensions, and how to verify them, most applications should choose to implement the handlers through a single handler type, which can have any number of dependencies injected such as keepers. In addition, this handler type could contain some notion of volatile vote extension state management which would assist in vote extension verification. This state management could be ephemeral or could be some form of on-disk persistence.

Example:

```
// VoteExtensionHandler implements an Oracle vote extension handler.
type VoteExtensionHandler struct {
```

```

cdc Codec
mk MyKeeper
state VoteExtState // This could be a map or a DB connection object
}

// ExtendVoteHandler can do something with h.mk and possibly h.state to create
// a vote extension, such as fetching a series of prices for supported assets.
func (h VoteExtensionHandler) ExtendVoteHandler(ctx sdk.Context, req
abci.RequestExtendVote) abci.ResponseExtendVote {
    prices := GetPrices(ctx, h.mk.Assets())
    bz, err := EncodePrices(h.cdc, prices)
    if err != nil {
        panic(fmt.Errorf("failed to encode prices for vote extension: %w", err))
    }

    // store our vote extension at the given height
    //
    // NOTE: Vote extensions can be overridden since we can timeout in a round.
    SetPrices(h.state, req, bz)

    return abci.ResponseExtendVote{VoteExtension: bz}
}

// VerifyVoteExtensionHandler can do something with h.state and req to verify
// the req.VoteExtension field, such as ensuring the provided oracle prices are
// within some valid range of our prices.
func (h VoteExtensionHandler) VerifyVoteExtensionHandler(ctx sdk.Context, req
abci.RequestVerifyVoteExtension) abci.ResponseVerifyVoteExtension {
    prices, err := DecodePrices(h.cdc, req.VoteExtension)
    if err != nil {
        log("failed to decode vote extension", "err", err)
        return abci.ResponseVerifyVoteExtension{Status: REJECT}
    }

    if err := ValidatePrices(h.state, req, prices); err != nil {
        log("failed to validate vote extension", "prices", prices, "err", err)
        return abci.ResponseVerifyVoteExtension{Status: REJECT}
    }

    // store updated vote extensions at the given height
    //
    // NOTE: Vote extensions can be overridden since we can timeout in a round.
    SetPrices(h.state, req, req.VoteExtension)

    return abci.ResponseVerifyVoteExtension{Status: ACCEPT}
}

```

Vote Extension Propagation & Verification

As mentioned previously, vote extensions for height H are only made available to the proposer at height $H+1$ during `PrepareProposal`. However, in order to make vote extensions useful, all validators should have access to the agreed upon vote extensions at height H during $H+1$.

Since CometBFT includes all the vote extension signatures in `RequestPrepareProposal`, we propose that the proposing validator manually "inject" the vote extensions along with their respective signatures via a special transaction, `VoteExtsTx`, into the block proposal during `PrepareProposal`. The `VoteExtsTx` will be populated with a single `ExtendedCommitInfo` object which is received directly from `RequestPrepareProposal`.

For convention, the `VoteExtsTx` transaction should be the first transaction in the block proposal, although chains can implement their own preferences. For safety purposes, we also propose that the proposer itself verify all the vote extension signatures it receives in `RequestPrepareProposal`.

A validator, upon a `RequestProcessProposal`, will receive the injected `VoteExtsTx` which includes the vote extensions along with their signatures. If no such transaction exists, the validator MUST REJECT the proposal.

When a validator inspects a `VoteExtsTx`, it will evaluate each `SignedVoteExtension`. For each signed vote extension, the validator will generate the signed bytes and verify the signature. At least 2/3 valid signatures, based on voting power, must be received in order for the block proposal to be valid, otherwise the validator MUST REJECT the proposal.

In order to have the ability to validate signatures, `BaseApp` must have access to the `x/staking` module, since this module stores an index from consensus address to public key. However, we will avoid a direct dependency on `x/staking` and instead rely on an interface instead. In addition, the Cosmos SDK will expose a default signature verification method which applications can use:

```
type ValidatorStore interface {
    GetValidatorByConsAddr(sdk.Context, cryptotypes.Address) (cryptotypes.PubKey,
        error)
}

// ValidateVoteExtensions is a function that an application can execute in
// ProcessProposal to verify vote extension signatures.
func (app *BaseApp) ValidateVoteExtensions(ctx sdk.Context, currentHeight int64,
extCommit abci.ExtendedCommitInfo) error {
    for _, vote := range extCommit.Votes {
        if !vote.SignedLastBlock || len(vote.VoteExtension) == 0 {
            continue
        }

        valConsAddr := cmtcrypto.Address(vote.Validator.Address)

        validator, err := app.validatorStore.GetValidatorByConsAddr(ctx,
valConsAddr)
        if err != nil {
            return fmt.Errorf("failed to get validator %s for vote extension",
valConsAddr)
        }

        cmtPubKey, err := validator.CmtConsPublicKey()
        if err != nil {
            return fmt.Errorf("failed to convert public key: %w", err)
        }
    }
}
```

```

    if len(vote.ExtensionSignature) == 0 {
        return fmt.Errorf("received a non-empty vote extension with empty
signature for validator %s", valConsAddr)
    }

    cve := cmtproto.CanonicalVoteExtension{
        Extension: vote.VoteExtension,
        Height:    currentHeight - 1, // the vote extension was signed in the
previous height
        Round:     int64(extCommit.Round),
        ChainId:   app.GetChainID(),
    }

    extSignBytes, err := cosmosio.MarshalDelimited(&cve)
    if err != nil {
        return fmt.Errorf("failed to encode CanonicalVoteExtension: %w", err)
    }

    if !cmtPubKey.VerifySignature(extSignBytes, vote.ExtensionSignature) {
        return errors.New("received vote with invalid signature")
    }

    return nil
}
}

```

Once at least 2/3 signatures, by voting power, are received and verified, the validator can use the vote extensions to derive additional data or come to some decision based on the vote extensions.

NOTE: It is very important to state, that neither the vote propagation technique nor the vote extension verification mechanism described above is required for applications to implement. In other words, a proposer is not required to verify and propagate vote extensions along with their signatures nor are proposers required to verify those signatures. An application can implement its own PKI mechanism and use that to sign and verify vote extensions.

Vote Extension Persistence

In certain contexts, it may be useful or necessary for applications to persist data derived from vote extensions. In order to facilitate this use case, we propose to allow app developers to define a pre-FinalizeBlock hook which will be called at the very beginning of `FinalizeBlock`, i.e. before `BeginBlock` (see below).

Note, we cannot allow applications to directly write to the application state during `ProcessProposal` because during replay, CometBFT will NOT call `ProcessProposal`, which would result in an incomplete state view.

```

func (a MyApp) PreFinalizeBlockHook(ctx sdk.Context, req.RequestFinalizeBlock) error
{
    voteExts := GetVoteExtensions(ctx, req.Txs)

    // Process and perform some compute on vote extensions, storing any resulting

```

```

// state.
if err = a.processVoteExtensions(ctx, voteExts); if err != nil {
    return err
}
}

```

FinalizeBlock

The existing ABCI methods `BeginBlock`, `DeliverTx`, and `EndBlock` have existed since the dawn of ABCI-based applications. Thus, applications, tooling, and developers have grown used to these methods and their use-cases. Specifically, `BeginBlock` and `EndBlock` have grown to be pretty integral and powerful within ABCI-based applications. E.g. an application might want to run distribution and inflation related operations prior to executing transactions and then have staking related changes to happen after executing all transactions.

We propose to keep `BeginBlock` and `EndBlock` within the SDK's core module interfaces only so application developers can continue to build against existing execution flows. However, we will remove `BeginBlock`, `DeliverTx` and `EndBlock` from the SDK's `BaseApp` implementation and thus the ABCI surface area.

What will then exist is a single `FinalizeBlock` execution flow. Specifically, in `FinalizeBlock` we will execute the application's `BeginBlock`, followed by execution of all the transactions, finally followed by execution of the application's `EndBlock`.

Note, we will still keep the existing transaction execution mechanics within `BaseApp`, but all notions of `DeliverTx` will be removed, i.e. `deliverState` will be replaced with `finalizeState`, which will be committed on `Commit`.

However, there are current parameters and fields that exist in the existing `BeginBlock` and `EndBlock` ABCI types, such as votes that are used in distribution and byzantine validators used in evidence handling. These parameters exist in the `FinalizeBlock` request type, and will need to be passed to the application's implementations of `BeginBlock` and `EndBlock`.

This means the Cosmos SDK's core module interfaces will need to be updated to reflect these parameters. The easiest and most straightforward way to achieve this is to just pass `RequestFinalizeBlock` to `BeginBlock` and `EndBlock`. Alternatively, we can create dedicated proxy types in the SDK that reflect these legacy ABCI types, e.g. `LegacyBeginBlockRequest` and `LegacyEndBlockRequest`. Or, we can come up with new types and names altogether.

```

func (app *BaseApp) FinalizeBlock(req abci.RequestFinalizeBlock)
(*abci.ResponseFinalizeBlock, error) {
    ctx := ...

    if app.preFinalizeBlockHook != nil {
        if err := app.preFinalizeBlockHook(ctx, req); err != nil {
            return nil, err
        }
    }

    beginBlockResp := app.beginBlock(ctx, req)
    appendBlockEventAttr(beginBlockResp.Events, "begin_block")
}

```

```

txExecResults := make([]abci.ExecTxResult, 0, len(req.Txs))
for _, tx := range req.Txs {
    result := app.runTx(runTxModeFinalize, tx)
    txExecResults = append(txExecResults, result)
}

endBlockResp := app.endBlock(ctx, req)
appendBlockEventAttr(beginBlockResp.Events, "end_block")

return abci.ResponseFinalizeBlock{
    TxResults:          txExecResults,
    Events:             joinEvents(beginBlockResp.Events,
endBlockResp.Events),
    ValidatorUpdates:   endBlockResp.ValidatorUpdates,
    ConsensusParamUpdates: endBlockResp.ConsensusParamUpdates,
    AppHash:            nil,
}
}

```

Events

Many tools, indexers and ecosystem libraries rely on the existence `BeginBlock` and `EndBlock` events. Since CometBFT now only exposes `FinalizeBlockEvents`, we find that it will still be useful for these clients and tools to still query for and rely on existing events, especially since applications will still define `BeginBlock` and `EndBlock` implementations.

In order to facilitate existing event functionality, we propose that all `BeginBlock` and `EndBlock` events have a dedicated `EventAttribute` with `key=block` and `value=begin_block|end_block`. The `EventAttribute` will be appended to each event in both `BeginBlock` and `EndBlock` events`.

Upgrading

CometBFT defines a consensus parameter, `VoteExtensionsEnableHeight`, which specifies the height at which vote extensions are enabled and **required**. If the value is set to zero, which is the default, then vote extensions are disabled and an application is not required to implement and use vote extensions.

However, if the value `H` is positive, at all heights greater than the configured height `H` vote extensions must be present (even if empty). When the configured height `H` is reached, `PrepareProposal` will not include vote extensions yet, but `ExtendVote` and `VerifyVoteExtension` will be called. Then, when reaching height `H+1`, `PrepareProposal` will include the vote extensions from height `H`.

It is very important to note, for all heights after `H`:

- Vote extensions CANNOT be disabled
- They are mandatory, i.e. all pre-commit messages sent MUST have an extension attached (even if empty)

When an application updates to the Cosmos SDK version with CometBFT v0.38 support, in the upgrade handler it must ensure to set the consensus parameter `VoteExtensionsEnableHeight` to the correct value. E.g. if an application is set to perform an upgrade at height `H`, then the value of

`VoteExtensionsEnableHeight` should be set to any value `>=H+1`. This means that at the upgrade height, `H`, vote extensions will not be enabled yet, but at height `H+1` they will be enabled.

Consequences

Backwards Compatibility

ABCI 2.0 is naturally not backwards compatible with prior versions of the Cosmos SDK and CometBFT. For example, an application that requests `RequestFinalizeBlock` to the same application that does not speak ABCI 2.0 will naturally fail.

In addition, `BeginBlock`, `DeliverTx` and `EndBlock` will be removed from the application ABCI interfaces and along with the inputs and outputs being modified in the module interfaces.

Positive

- `BeginBlock` and `EndBlock` semantics remain, so burden on application developers should be limited.
- Less communication overhead as multiple ABCI requests are condensed into a single request.
- Sets the groundwork for optimistic execution.
- Vote extensions allow for an entirely new set of application primitives to be developed, such as in-process price oracles and encrypted mempools.

Negative

- Some existing Cosmos SDK core APIs may need to be modified and thus broken.
- Signature verification in `ProcessProposal` of 100+ vote extension signatures will add significant performance overhead to `ProcessProposal`. Granted, the signature verification process can happen concurrently using an error group with `GOMAXPROCS` goroutines.

Neutral

- Having to manually "inject" vote extensions into the block proposal during `PrepareProposal` is an awkward approach and takes up block space unnecessarily.
- The requirement of `ResetProcessProposalState` can create a footgun for application developers if they're not careful, but this is necessary in order for applications to be able to commit state from vote extension computation.

Further Discussions

Future discussions include design and implementation of ABCI 3.0, which is a continuation of ABCI++ and the general discussion of optimistic execution.

References

- [ADR 060: ABCI 1.0 \(Phase I\)](#)

ADR-065: Store V2

Changelog

- Feb 14, 2023: Initial Draft (@alexanderbez)

Status

DRAFT

Abstract

The storage and state primitives that Cosmos SDK based applications have used have by and large not changed since the launch of the inaugural Cosmos Hub. The demands and needs of Cosmos SDK based applications, from both developer and client UX perspectives, have evolved and outgrown the ecosystem since these primitives were first introduced.

Over time as these applications have gained significant adoption, many critical shortcomings and flaws have been exposed in the state and storage primitives of the Cosmos SDK.

In order to keep up with the evolving demands and needs of both clients and developers, a major overhaul to these primitives are necessary.

Context

The Cosmos SDK provides application developers with various storage primitives for dealing with application state. Specifically, each module contains its own merkle commitment data structure -- an IAVL tree. In this data structure, a module can store and retrieve key-value pairs along with Merkle commitments, i.e. proofs, to those key-value pairs indicating that they do or do not exist in the global application state. This data structure is the base layer `KVStore`.

In addition, the SDK provides abstractions on top of this Merkle data structure. Namely, a root multi-store (RMS) is a collection of each module's `KVStore`. Through the RMS, the application can serve queries and provide proofs to clients in addition to provide a module access to its own unique `KVStore` though the use of `StoreKey`, which is an OCAP primitive.

There are further layers of abstraction that sit between the RMS and the underlying IAVL `KVStore`. A `GasKVStore` is responsible for tracking gas IO consumption for state machine reads and writes. A `CacheKVStore` is responsible for providing a way to cache reads and buffer writes to make state transitions atomic, e.g. transaction execution or governance proposal execution.

There are a few critical drawbacks to these layers of abstraction and the overall design of storage in the Cosmos SDK:

- Since each module has its own IAVL `KVStore`, commitments are not atomic
 - Note, we can still allow modules to have their own IAVL `KVStore`, but the IAVL library will need to support the ability to pass a DB instance as an argument to various IAVL APIs.
- Since IAVL is responsible for both state storage and commitment, running an archive node becomes increasingly expensive as disk space grows exponentially.
- As the size of a network increases, various performance bottlenecks start to emerge in many areas such as query performance, network upgrades, state migrations, and general application performance.
- Developer UX is poor as it does not allow application developers to experiment with different types of approaches to storage and commitments, along with the complications of many layers of abstractions referenced above.

See the [Storage Discussion](#) for more information.

Alternatives

There was a previous attempt to refactor the storage layer described in [ADR-040](#). However, this approach mainly stems on the short comings of IAVL and various performance issues around it. While there was a (partial) implementation of [ADR-040](#), it was never adopted for a variety of reasons, such as the reliance on using an SMT, which was more in a research phase, and some design choices that couldn't be fully agreed upon, such as the snap-shutting mechanism that would result in massive state bloat.

Decision

We propose to build upon some of the great ideas introduced in [ADR-040](#), while being a bit more flexible with the underlying implementations and overall less intrusive. Specifically, we propose to:

- Separate the concerns of state commitment (**SC**), needed for consensus, and state storage (**SS**), needed for state machine and clients.
- Reduce layers of abstractions necessary between the RMS and underlying stores.
- Provide atomic module store commitments by providing a batch database object to core IAVL APIs.
- Reduce complexities in the `CacheKVStore` implementation while also improving performance^[3].

Furthermore, we will keep the IAVL is the backing [commitment](#) store for the time being. While we might not fully settle on the use of IAVL in the long term, we do not have strong empirical evidence to suggest a better alternative. Given that the SDK provides interfaces for stores, it should be sufficient to change the backing commitment store in the future should evidence arise to warrant a better alternative. However there is promising work being done to IAVL that should result in significant performance improvement [1,2].

Separating SS and SC

By separating SS and SC, it will allow for us to optimize against primary use cases and access patterns to state. Specifically, The SS layer will be responsible for direct access to data in the form of (key, value) pairs, whereas the SC layer (IAVL) will be responsible for committing to data and providing Merkle proofs.

Note, the underlying physical storage database will be the same between both the SS and SC layers. So to avoid collisions between (key, value) pairs, both layers will be namespaced.

State Commitment (SC)

Given that the existing solution today acts as both SS and SC, we can simply repurpose it to act solely as the SC layer without any significant changes to access patterns or behavior. In other words, the entire collection of existing IAVL-backed module `KVStore`s will act as the SC layer.

However, in order for the SC layer to remain lightweight and not duplicate a majority of the data held in the SS layer, we encourage node operators to keep tight pruning strategies.

State Storage (SS)

In the RMS, we will expose a *single* `KVStore` backed by the same physical database that backs the SC layer. This `KVStore` will be explicitly namespaced to avoid collisions and will act as the primary storage for (key, value) pairs.

While we most likely will continue the use of `cosmos-db`, or some local interface, to allow for flexibility and iteration over preferred physical storage backends as research and benchmarking continues. However, we propose to hardcode the use of RocksDB as the primary physical storage backend.

Since the SS layer will be implemented as a `KVStore`, it will support the following functionality:

- Range queries
- CRUD operations
- Historical queries and versioning
- Pruning

The RMS will keep track of all buffered writes using a dedicated and internal `MemoryListener` for each `StoreKey`. For each block height, upon `Commit`, the SS layer will write all buffered (key, value) pairs under a [RocksDB user-defined timestamp](#) column family using the block height as the timestamp, which is an unsigned integer. This will allow a client to fetch (key, value) pairs at historical and current heights along with making iteration and range queries relatively performant as the timestamp is the key suffix.

Note, we choose not to use a more general approach of allowing any embedded key/value database, such as LevelDB or PebbleDB, using height key-prefixed keys to effectively version state because most of these databases use variable length keys which would effectively make actions like iteration and range queries less performant.

Since operators might want pruning strategies to differ in SS compared to SC, e.g. having a very tight pruning strategy in SC while having a looser pruning strategy for SS, we propose to introduce an additional pruning configuration, with parameters that are identical to what exists in the SDK today, and allow operators to control the pruning strategy of the SS layer independently of the SC layer.

Note, the SC pruning strategy must be congruent with the operator's state sync configuration. This is so as to allow state sync snapshots to execute successfully, otherwise, a snapshot could be triggered on a height that is not available in SC.

State Sync

The state sync process should be largely unaffected by the separation of the SC and SS layers. However, if a node syncs via state sync, the SS layer of the node will not have the state synced height available, since the IAVL import process is not setup in a way to easily allow direct key/value insertion. A modification of the IAVL import process would be necessary to facilitate having the state sync height available.

Note, this is not problematic for the state machine itself because when a query is made, the RMS will automatically direct the query correctly (see [Queries](#)).

Queries

To consolidate the query routing between both the SC and SS layers, we propose to have a notion of a "query router" that is constructed in the RMS. This query router will be supplied to each `KVStore` implementation. The query router will route queries to either the SC layer or the SS layer based on a few parameters. If `prove: true`, then the query must be routed to the SC layer. Otherwise, if the query height is available in the SS layer, the query will be served from the SS layer. Otherwise, we fall back on the SC layer.

If no height is provided, the SS layer will assume the latest height. The SS layer will store a reverse index to lookup `LatestVersion -> timestamp(version)` which is set on `Commit`.

Proofs

Since the SS layer is naturally a storage layer only, without any commitments to (key, value) pairs, it cannot provide Merkle proofs to clients during queries.

Since the pruning strategy against the SC layer is configured by the operator, we can therefore have the RMS route the query SC layer if the version exists and `prove: true`. Otherwise, the query will fall back to the SS layer without a proof.

We could explore the idea of using state snapshots to rebuild an in-memory IAVL tree in real time against a version closest to the one provided in the query. However, it is not clear what the performance implications will be of this approach.

Atomic Commitment

We propose to modify the existing IAVL APIs to accept a batch DB object instead of relying on an internal batch object in `nodeDB`. Since each underlying IAVL `KVStore` shares the same DB in the SC layer, this will allow commits to be atomic.

Specifically, we propose to:

- Remove the `dbm.Batch` field from `nodeDB`
- Update the `SaveVersion` method of the `MutableTree` IAVL type to accept a batch object
- Update the `Commit` method of the `CommitKVStore` interface to accept a batch object
- Create a batch object in the RMS during `Commit` and pass this object to each `KVStore`
- Write the database batch after all stores have committed successfully

Note, this will require IAVL to be updated to not rely or assume on any batch being present during `SaveVersion`.

Consequences

As a result of a new store V2 package, we should expect to see improved performance for queries and transactions due to the separation of concerns. We should also expect to see improved developer UX around experimentation of commitment schemes and storage backends for further performance, in addition to a reduced amount of abstraction around KVStores making operations such as caching and state branching more intuitive.

However, due to the proposed design, there are drawbacks around providing state proofs for historical queries.

Backwards Compatibility

This ADR proposes changes to the storage implementation in the Cosmos SDK through an entirely new package. Interfaces may be borrowed and extended from existing types that exist in `store`, but no existing implementations or interfaces will be broken or modified.

Positive

- Improved performance of independent SS and SC layers
- Reduced layers of abstraction making storage primitives easier to understand
- Atomic commitments for SC
- Redesign of storage types and interfaces will allow for greater experimentation such as different physical storage backends and different commitment schemes for different application modules

Negative

- Providing proofs for historical state is challenging

Neutral

- Keeping IAVL as the primary commitment data structure, although drastic performance improvements are being made

Further Discussions

Module Storage Control

Many modules store secondary indexes that are typically solely used to support client queries, but are actually not needed for the state machine's state transitions. What this means is that these indexes technically have no reason to exist in the SC layer at all, as they take up unnecessary space. It is worth exploring what an API would look like to allow modules to indicate what (key, value) pairs they want to be persisted in the SC layer, implicitly indicating the SS layer as well, as opposed to just persisting the (key, value) pair only in the SS layer.

Historical State Proofs

It is not clear what the importance or demand is within the community of providing commitment proofs for historical state. While solutions can be devised such as rebuilding trees on the fly based on state snapshots, it is not clear what the performance implications are for such solutions.

Physical DB Backends

This ADR proposes usage of RocksDB to utilize user-defined timestamps as a versioning mechanism. However, other physical DB backends are available that may offer alternative ways to implement versioning while also providing performance improvements over RocksDB. E.g. PebbleDB supports MVCC timestamps as well, but we'll need to explore how PebbleDB handles compaction and state growth over time.

References

- [1] <https://github.com/cosmos/ibc/pull/676>
- [2] <https://github.com/cosmos/ibc/pull/664>
- [3] <https://github.com/cosmos/cosmos-sdk/issues/14990>

ADR 067: Simulator v2

Changelog

- June 01, 2023: Initial Draft (@alexanderbez)

Status

DRAFT

Abstract

The Cosmos SDK simulator is a tool that allows developers to test the entirety of their application's state machine through the use of pseudo-randomized "operations", which represent transactions. The simulator also provides primitives that ensure there are no non-determinism issues and that the application's state machine can be successfully exported and imported using randomized state.

The simulator has played an absolutely critical role in the development and testing of the Cosmos Hub and all the releases of the Cosmos SDK after the launch of the Cosmos Hub. Since the Hub, the simulator has relatively not changed much, so it's overdue for a revamp.

Context

The current simulator, `x/simulation`, acts as a semi-fuzz testing suite that takes in an integer that represents a seed into a PRNG. The PRNG is used to generate a sequence of "operations" that are meant to reflect transactions that an application's state machine can process. Through the use of the PRNG, all aspects of block production and consumption are randomized. This includes a block's proposer, the validators who both sign and miss the block, along with the transaction operations themselves.

Each Cosmos SDK module defines a set of simulation operations that *attempt* to produce valid transactions, e.g. `x/distribution/simulation/operations.go`. These operations can sometimes fail depending on the accumulated state of the application within that simulation run. The simulator will continue to generate operations until it has reached a certain number of operations or until it has reached a fatal state, reporting results. This gives the ability for application developers to reliably execute full range application simulation and fuzz testing against their application.

However, there are a few major drawbacks. Namely, with the advent of ABCI++, specifically `FinalizeBlock`, the internal workings of the simulator no longer comply with how an application would actually perform. Specifically, operations are executed *after* `FinalizeBlock`, whereas they should be executed *within* `FinalizeBlock`.

Additionally, the simulator is not very extensible. Developers should be able to easily define and extend the following:

- Consistency or validity predicates (what are known as invariants today)
- Property tests of state before and after a block is simulated

In addition, we also want to achieve the following:

- Consolidated weight management, i.e. define weights within the simulator itself via a config and not defined in each module
- Observability of the simulator's execution, i.e. have easy to understand output/logs with the ability to pipe those logs into some external sink
- Smart replay, i.e. the ability to not only rerun a simulation from a seed, but also the ability to replay from an arbitrary breakpoint
- Run a simulation based off of real network state

Decision

Instead of refactoring the existing simulator, `x/simulation`, we propose to create a new package in the root of the Cosmos SDK, `simulator`, that will be the new simulation framework. The simulator will more accurately reflect the complete lifecycle of an ABCI application.

Specifically, we propose a similar implementation and use of a `simulator.Manager`, that exists today, that is responsible for managing the execution of a simulation. The manager will wrap an ABCI application and will be responsible for the following:

- Populating the application's mempool with a set of pseudo-random transactions before each block, some of which may contain invalid messages.
- Selecting transactions and a random proposer to execute `PrepareProposal`.
- Executing `ProcessProposal`, `FinalizeBlock` and `Commit`.
- Executing a set of validity predicates before and after each block.
- Maintaining a CPU and RAM profile of the simulation execution.
- Allowing a simulation to stop and resume from a given block height.

- Simulation liveness of each validator per-block.

From an application developer's perspective, they will only need to provide the modules to be used in the simulator and the manager will take care of the rest. In addition, they will not need to write their own simulation test(s), e.g. non-determinism, multi-seed, etc..., as the manager will provide these as well.

```
type Manager struct {
    app      sdk.Application
    mempool sdk.Mempool
    rng      rand.Rand
    // ...
}
```

Configuration

The simulator's testing input will be driven by a configuration file, as opposed to CLI arguments. This will allow for more extensibility and ease of use along with the ability to have shared configuration files across multiple simulations.

Execution

As alluded to previously, after the execution of each block, the manager will generate a series of pseudo-random transactions and attempt to insert them into the mempool via `BaseApp#CheckTx`. During the ABCI lifecycle of a block, this mempool will be used to seed the transactions into a block proposal as it would in a real network. This allows us to not only test the state machine, but also test the ABCI lifecycle of a block.

Statistics, such as total blocks and total failed proposals, will be collected, logged and written to output after the full or partial execution of a simulation. The output destination of these statistics will be configurable.

```
func (s *Simulator) SimulateBlock() {
    rProposer := s.SelectRandomProposer()
    rTxs := s.SelectTxs()

    prepareResp, err := s.app.PrepareProposal(&abci.RequestPrepareProposal{Txs: rTxs})
    // handle error

    processResp, err := s.app.ProcessProposal(&abci.RequestProcessProposal{
        Txs: prepareResp.Txs,
        // ...
    })
    // handle error

    // execute liveness matrix...

    _, err = s.app.FinalizeBlock(...)
    // handle error

    _, err = s.app.Commit(...)
    // handle error
}
```

Note, some application do not define or need their own app-side mempool, so we propose that `SelectTxs` mimic CometBFT and just return FIFO-ordered transactions from an ad-hoc simulator mempool. In the case where an application does define its own mempool, it will simply ignore what is provided in `RequestPrepareProposal`.

Profiling

The manager will be responsible for collecting CPU and RAM profiles of the simulation execution. We propose to use [Pyroscope](#) to capture profiles and export them to a local file and via an HTTP endpoint. This can be disabled or enabled by configuration.

Breakpoints

Via configuration, a caller can express a height-based breakpoint that will allow the simulation to stop and resume from a given height. This will allow for debugging of CPU, RAM, and state.

Validity Predicates

We propose to provide the ability for an application to provide the simulator a set of validity predicates, i.e. invariant checkers, that will be executed before and after each block. This will allow for the application to assert that certain state invariants are held before and after each block. Note, as a consequence of this, we propose to remove the existing notion of invariants from module production execution paths and deprecate their usage all together.

```
type Manager struct {
    // ...
    preBlockValidator func(sdk.Context) error
    postBlockValidator func(sdk.Context) error
}
```

Consequences

Backwards Compatibility

The new simulator package will not naturally not be backwards compatible with the existing `x/simulation` module. However, modules will still be responsible for providing pseudo-random transactions to the simulator.

Positive

- Providing more intuitive and cleaner APIs for application developers
- More closely resembling the true lifecycle of an ABCI application

Negative

- Breaking current Cosmos SDK module APIs for transaction generation

References

- [Osmosis Simulation ADR](#)

ADR {ADR-NUMBER}: {TITLE}

Changelog

- {date}: {changelog}

Status

{DRAFT | PROPOSED} Not Implemented

Please have a look at the [PROCESS](#) page. Use DRAFT if the ADR is in a draft stage (draft PR) or PROPOSED if it's in review.

Abstract

"If you can't explain it simply, you don't understand it well enough." Provide a simplified and layman-accessible explanation of the ADR. A short (~200 word) description of the issue being addressed.

Context

This section describes the forces at play, including technological, political, social, and project local. These forces are probably in tension, and should be called out as such. The language in this section is value-neutral. It is simply describing facts. It should clearly explain the problem and motivation that the proposal aims to resolve. {context body}

Alternatives

This section describes alternative designs to the chosen design. This section is important and if an adr does not have any alternatives then it should be considered that the ADR was not thought through.

Decision

This section describes our response to these forces. It is stated in full sentences, with active voice. "We will ..." {decision body}

Consequences

This section describes the resulting context, after applying the decision. All consequences should be listed here, not just the "positive" ones. A particular decision may have positive, negative, and neutral consequences, but all of them affect the team and project in the future.

Backwards Compatibility

All ADRs that introduce backwards incompatibilities must include a section describing these incompatibilities and their severity. The ADR must explain how the author proposes to deal with these incompatibilities. ADR submissions without a sufficient backwards compatibility treatise may be rejected outright.

Positive

{positive consequences}

Negative

{negative consequences}

Neutral

{neutral consequences}

Further Discussions

While an ADR is in the DRAFT or PROPOSED stage, this section should contain a summary of issues to be solved in future iterations (usually referencing comments from a pull-request discussion).

Later, this section can optionally list ideas or improvements the author or reviewers found during the analysis of this ADR.

Test Cases [optional]

Test cases for an implementation are mandatory for ADRs that are affecting consensus changes. Other ADRs can choose to include links to test cases if applicable.

References

- {reference link}
-

title: Cosmos SDK Documentation sidebar_position: 0

Cosmos SDK is the world's most popular framework for building application-specific blockchains.

Getting Started

Read all about the SDK or dive straight into the code with tutorials.

- [Introductions to the Cosmos SDK](#) - Learn about all the parts of the Cosmos SDK.
- [SDK Tutorials](#) - Build a complete blockchain application from scratch.

Explore the SDK

Get familiar with the SDK and explore its main concepts.

- [Introduction](#) - High-level overview of the Cosmos SDK.
- [Basics](#) - Anatomy of a blockchain, transaction lifecycle, accounts and more.
- [Core Concepts](#) - Read about the core concepts like baseapp, the store, or the server.
- [Building Modules](#) - Discover how to build modules for the Cosmos SDK.
- [Running a Node](#) - Running and interacting with nodes using the CLI and API.
- [Modules](#) - Explore existing modules to build your application with.

Explore the Stack

Check out the docs for the various parts of the Cosmos stack.

- [Cosmos Hub](#) - The first of thousands of interconnected blockchains on the Cosmos Network.
- [CometBFT](#) - The leading BFT engine for building blockchains, powering Cosmos SDK.

Help & Support

- [GitHub Discussions](#) - Ask questions and discuss SDK development on GitHub.
- [Discord](#) - Chat with Cosmos developers on Discord.
- [Found an issue?](#) - Help us improve this page by suggesting edits on GitHub.

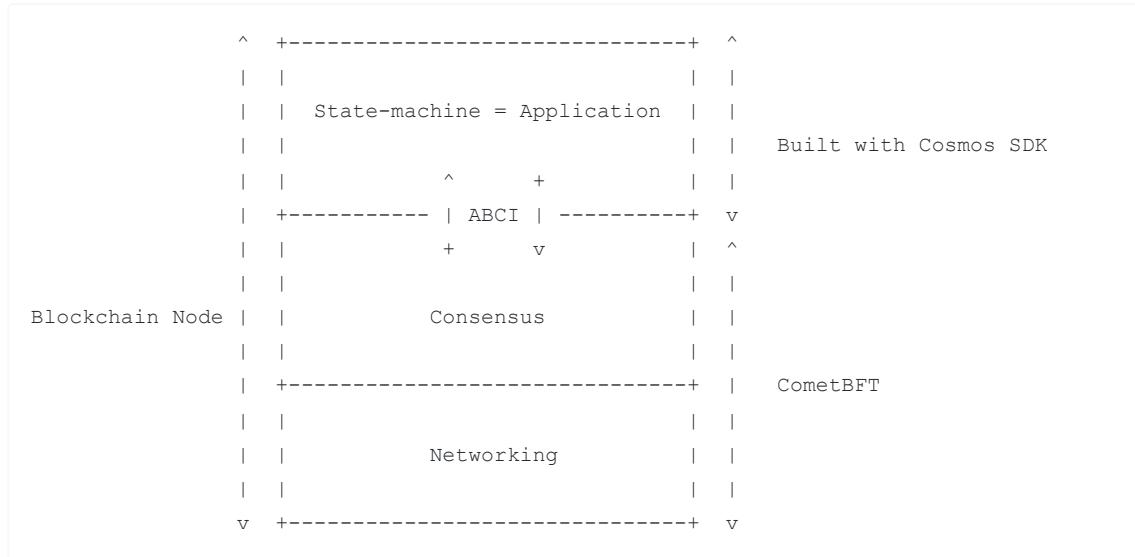
sidebar_position: 1

Anatomy of a Cosmos SDK Application

:::note Synopsis This document describes the core parts of a Cosmos SDK application, represented throughout the document as a placeholder application named `app` . :::

Node Client

The Daemon, or [Full-Node Client](#), is the core process of a Cosmos SDK-based blockchain. Participants in the network run this process to initialize their state-machine, connect with other full-nodes, and update their state-machine as new blocks come in.



The blockchain full-node presents itself as a binary, generally suffixed by `-d` for "daemon" (e.g. `appd` for `app` or `gaiad` for `gaia`). This binary is built by running a simple [`main.go`](#) function placed in `./cmd/appd/`. This operation usually happens through the [Makefile](#).

Once the main binary is built, the node can be started by running the [`start command`](#). This command function primarily does three things:

1. Create an instance of the state-machine defined in [`app.go`](#) .
2. Initialize the state-machine with the latest known state, extracted from the `db` stored in the `~/app/data` folder. At this point, the state-machine is at height `appBlockHeight` .
3. Create and start a new CometBFT instance. Among other things, the node performs a handshake with its peers. It gets the latest `blockHeight` from them and replays blocks to sync to this height

if it is greater than the local `appBlockHeight`. The node starts from genesis and CometBFT sends an `InitChain` message via the ABCI to the `app`, which triggers the [InitChainer](#).

:::note When starting a CometBFT instance, the genesis file is the `0` height and the state within the genesis file is committed at block height `1`. When querying the state of the node, querying block height `0` will return an error. :::

Core Application File

In general, the core of the state-machine is defined in a file called `app.go`. This file mainly contains the **type definition of the application** and functions to **create and initialize it**.

Type Definition of the Application

The first thing defined in `app.go` is the `type` of the application. It is generally comprised of the following parts:

- A reference to [baseapp](#). The custom application defined in `app.go` is an extension of `baseapp`. When a transaction is relayed by CometBFT to the application, `app` uses `baseapp`'s methods to route them to the appropriate module. `baseapp` implements most of the core logic for the application, including all the [ABCI methods](#) and the [routing logic](#).
- A list of store keys. The [store](#), which contains the entire state, is implemented as a [multistore](#) (i.e. a store of stores) in the Cosmos SDK. Each module uses one or multiple stores in the multistore to persist their part of the state. These stores can be accessed with specific keys that are declared in the `app` type. These keys, along with the `keepers`, are at the heart of the [object-capabilities model](#) of the Cosmos SDK.
- A list of module's keepers. Each module defines an abstraction called [keeper](#), which handles reads and writes for this module's store(s). The `keeper`'s methods of one module can be called from other modules (if authorized), which is why they are declared in the application's type and exported as interfaces to other modules so that the latter can only access the authorized functions.
- A reference to an [appCodec](#). The application's `appCodec` is used to serialize and deserialize data structures in order to store them, as stores can only persist `[]bytes`. The default codec is [Protocol Buffers](#).
- A reference to a [legacyAmino](#) codec. Some parts of the Cosmos SDK have not been migrated to use the `appCodec` above, and are still hardcoded to use Amino. Other parts explicitly use Amino for backwards compatibility. For these reasons, the application still holds a reference to the legacy Amino codec. Please note that the Amino codec will be removed from the SDK in the upcoming releases.
- A reference to a [module manager](#) and a [basic module manager](#). The module manager is an object that contains a list of the application's modules. It facilitates operations related to these modules, like registering their [Msg service](#) and [gRPC Query service](#), or setting the order of execution between modules for various functions like [InitChainer](#), [BeginBlocker and EndBlocker](#).

See an example of application type definition from `simapp`, the Cosmos SDK's own app used for demo and testing purposes:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L173-L212
```

Constructor Function

Also defined in `app.go` is the constructor function, which constructs a new application of the type defined in the preceding section. The function must fulfill the `AppCreator` signature in order to be used in the [start command](#) of the application's daemon command.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/types/app.go#L66-L68
```

Here are the main actions performed by this function:

- Instantiate a new `codec` and initialize the `codec` of each of the application's modules using the [basic manager](#).
- Instantiate a new application with a reference to a `baseapp` instance, a codec, and all the appropriate store keys.
- Instantiate all the `keeper` objects defined in the application's `type` using the `NewKeeper` function of each of the application's modules. Note that keepers must be instantiated in the correct order, as the `NewKeeper` of one module might require a reference to another module's `keeper`.
- Instantiate the application's [module manager](#) with the `AppModule` object of each of the application's modules.
- With the module manager, initialize the application's [Msg services](#), [gRPC Query services](#), [legacy Msg routes](#), and [legacy query routes](#). When a transaction is relayed to the application by CometBFT via the ABCI, it is routed to the appropriate module's `Msg service` using the routes defined here. Likewise, when a gRPC query request is received by the application, it is routed to the appropriate module's `gRPC query service` using the gRPC routes defined here. The Cosmos SDK still supports legacy `Msg`s and legacy CometBFT queries, which are routed using the legacy `Msg` routes and the legacy query routes, respectively.
- With the module manager, register the [application's modules' invariants](#). Invariants are variables (e.g. total supply of a token) that are evaluated at the end of each block. The process of checking invariants is done via a special module called the `InvariantsRegistry`. The value of the invariant should be equal to a predicted value defined in the module. Should the value be different than the predicted one, special logic defined in the invariant registry is triggered (usually the chain is halted). This is useful to make sure that no critical bug goes unnoticed, producing long-lasting effects that are hard to fix.
- With the module manager, set the order of execution between the `InitGenesis`, `BeginBlocker`, and `EndBlocker` functions of each of the [application's modules](#). Note that not all modules implement these functions.
- Set the remaining application parameters:
 - `InitChainer` : used to initialize the application when it is first started.
 - `BeginBlocker` , `EndBlocker` : called at the beginning and at the end of every block.
 - `anteHandler` : used to handle fees and signature verification.
- Mount the stores.
- Return the application.

Note that the constructor function only creates an instance of the app, while the actual state is either carried over from the `~/.app/data` folder if the node is restarted, or generated from the genesis file if the node is started for the first time.

See an example of application constructor from `simapp` :

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L223-L575
```

InitChainer

The `InitChainer` is a function that initializes the state of the application from a genesis file (i.e. token balances of genesis accounts). It is called when the application receives the `InitChain` message from the CometBFT engine, which happens when the node is started at `appBlockHeight == 0` (i.e. on genesis). The application must set the `InitChainer` in its [constructor](#) via the [SetInitChainer](#) method.

In general, the `InitChainer` is mostly composed of the [InitGenesis](#) function of each of the application's modules. This is done by calling the `InitGenesis` function of the module manager, which in turn calls the `InitGenesis` function of each of the modules it contains. Note that the order in which the modules' `InitGenesis` functions must be called has to be set in the module manager using the [module manager's SetOrderInitGenesis](#) method. This is done in the [application's constructor](#), and the `SetOrderInitGenesis` has to be called before the `SetInitChainer`.

See an example of an `InitChainer` from `simapp`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L626-L634
```

BeginBlocker and EndBlocker

The Cosmos SDK offers developers the possibility to implement automatic execution of code as part of their application. This is implemented through two functions called `BeginBlocker` and `EndBlocker`. They are called when the application receives the `FinalizeBlock` messages from the CometBFT consensus engine, which happens respectively at the beginning and at the end of each block. The application must set the `BeginBlocker` and `EndBlocker` in its [constructor](#) via the [SetBeginBlocker](#) and [SetEndBlocker](#) methods.

In general, the `BeginBlocker` and `EndBlocker` functions are mostly composed of the [BeginBlock and EndBlock](#) functions of each of the application's modules. This is done by calling the `BeginBlock` and `EndBlock` functions of the module manager, which in turn calls the `BeginBlock` and `EndBlock` functions of each of the modules it contains. Note that the order in which the modules' `BeginBlock` and `EndBlock` functions must be called has to be set in the module manager using the `SetOrderBeginBlockers` and `SetOrderEndBlockers` methods, respectively. This is done via the [module manager](#) in the [application's constructor](#), and the `SetOrderBeginBlockers` and `SetOrderEndBlockers` methods have to be called before the `SetBeginBlocker` and `SetEndBlocker` functions.

As a sidenote, it is important to remember that application-specific blockchains are deterministic. Developers must be careful not to introduce non-determinism in `BeginBlocker` or `EndBlocker`, and must also be careful not to make them too computationally expensive, as [gas](#) does not constrain the cost of `BeginBlocker` and `EndBlocker` execution.

See an example of `BeginBlocker` and `EndBlocker` functions from `simapp`

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L613-L620
```

Register Codec

The `EncodingConfig` structure is the last important part of the `app.go` file. The goal of this structure is to define the codecs that will be used throughout the app.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/params/encoding.go#L9-L16
```

Here are descriptions of what each of the four fields means:

- `InterfaceRegistry` : The `InterfaceRegistry` is used by the Protobuf codec to handle interfaces that are encoded and decoded (we also say "unpacked") using `google.protobuf.Any`. `Any` could be thought as a struct that contains a `type_url` (name of a concrete type implementing the interface) and a `value` (its encoded bytes). `InterfaceRegistry` provides a mechanism for registering interfaces and implementations that can be safely unpacked from `Any`. Each application module implements the `RegisterInterfaces` method that can be used to register the module's own interfaces and implementations.
 - You can read more about `Any` in [ADR-019](#).
 - To go more into details, the Cosmos SDK uses an implementation of the Protobuf specification called `gogoproto`. By default, the [gogo protobuf implementation of Any](#) uses [global type registration](#) to decode values packed in `Any` into concrete Go types. This introduces a vulnerability where any malicious module in the dependency tree could register a type with the global protobuf registry and cause it to be loaded and unmarshaled by a transaction that referenced it in the `type_url` field. For more information, please refer to [ADR-019](#).
- `Codec` : The default codec used throughout the Cosmos SDK. It is composed of a `BinaryCodec` used to encode and decode state, and a `JSONCodec` used to output data to the users (for example, in the [CLI](#)). By default, the SDK uses Protobuf as `Codec`.
- `TxConfig` : `TxConfig` defines an interface a client can utilize to generate an application-defined concrete transaction type. Currently, the SDK handles two transaction types: `SIGN_MODE_DIRECT` (which uses Protobuf binary as over-the-wire encoding) and `SIGN_MODE_LEGACY_AMINO_JSON` (which depends on Amino). Read more about transactions [here](#).
- `Amino` : Some legacy parts of the Cosmos SDK still use Amino for backwards-compatibility. Each module exposes a `RegisterLegacyAmino` method to register the module's specific types within Amino. This `Amino` codec should not be used by app developers anymore, and will be removed in future releases.

An application should create its own encoding config. See an example of a `simappparams.EncodingConfig` from `simapp`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/params/encoding.go#L11-L16
```

Modules

[Modules](#) are the heart and soul of Cosmos SDK applications. They can be considered as state-machines nested within the state-machine. When a transaction is relayed from the underlying CometBFT engine via the ABCI to the application, it is routed by `baseapp` to the appropriate module in order to be processed.

This paradigm enables developers to easily build complex state-machines, as most of the modules they need often already exist. **For developers, most of the work involved in building a Cosmos SDK application revolves around building custom modules required by their application that do not exist yet, and integrating them with modules that do already exist into one coherent application.** In the application directory, the standard practice is to store modules in the `x/` folder (not to be confused with the Cosmos SDK's `x/` folder, which contains already-built modules).

Application Module Interface

Modules must implement [Interfaces](#) defined in the Cosmos SDK, [AppModuleBasic](#) and [AppModule](#). The former implements basic non-dependent elements of the module, such as the `codec`, while the latter handles the bulk of the module methods (including methods that require references to other modules' keepers). Both the `AppModule` and `AppModuleBasic` types are, by convention, defined in a file called `module.go`.

`AppModule` exposes a collection of useful methods on the module that facilitates the composition of modules into a coherent application. These methods are called from the [module manager](#), which manages the application's collection of modules.

Msg Services

Each application module defines two [Protobuf services](#): one `Msg` service to handle messages, and one gRPC `Query` service to handle queries. If we consider the module as a state-machine, then a `Msg` service is a set of state transition RPC methods. Each Protobuf `Msg` service method is 1:1 related to a Protobuf request type, which must implement `sdk.Msg` interface. Note that `sdk.Msg`s are bundled in [transactions](#), and each transaction contains one or multiple messages.

When a valid block of transactions is received by the full-node, CometBFT relays each one to the application via [DeliverTx](#). Then, the application handles the transaction:

1. Upon receiving the transaction, the application first unmarshalls it from `[]byte`.
2. Then, it verifies a few things about the transaction like [fee payment and signatures](#) before extracting the `Msg`(s) contained in the transaction.
3. `sdk.Msg`s are encoded using Protobuf [Any](#)s. By analyzing each `Any`'s `type_url`, baseapp's `msgServiceRouter` routes the `sdk.Msg` to the corresponding module's `Msg` service.
4. If the message is successfully processed, the state is updated.

For more details, see [transaction lifecycle](#).

Module developers create custom `Msg` services when they build their own module. The general practice is to define the `Msg` Protobuf service in a `tx.proto` file. For example, the `x/bank` module defines a service with two methods to transfer tokens:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/bank/v1beta1/tx.proto#L13-L36
```

Service methods use `keeper` in order to update the module state.

Each module should also implement the `RegisterServices` method as part of the [AppModule interface](#). This method should call the `RegisterMsgServer` function provided by the generated Protobuf code.

gRPC Query Services

gRPC `Query` services allow users to query the state using [gRPC](#). They are enabled by default, and can be configured under the `grpc.enable` and `grpc.address` fields inside [`app.toml`](#).

gRPC `Query` services are defined in the module's Protobuf definition files, specifically inside `query.proto`. The `query.proto` definition file exposes a single `Query` [Protobuf service](#). Each gRPC query endpoint corresponds to a service method, starting with the `rpc` keyword, inside the `Query` service.

Protobuf generates a `QueryServer` interface for each module, containing all the service methods. A module's `keeper` then needs to implement this `QueryServer` interface, by providing the concrete implementation of each service method. This concrete implementation is the handler of the corresponding gRPC query endpoint.

Finally, each module should also implement the `RegisterServices` method as part of the [AppModule interface](#). This method should call the `RegisterQueryServer` function provided by the generated Protobuf code.

Keeper

`Keepers` are the gatekeepers of their module's store(s). To read or write in a module's store, it is mandatory to go through one of its `keeper`'s methods. This is ensured by the [object-capabilities](#) model of the Cosmos SDK. Only objects that hold the key to a store can access it, and only the module's `keeper` should hold the key(s) to the module's store(s).

`Keepers` are generally defined in a file called `keeper.go`. It contains the `keeper`'s type definition and methods.

The `keeper` type definition generally consists of the following:

- **Key(s)** to the module's store(s) in the multistore.
- Reference to **other module's keepers**. Only needed if the `keeper` needs to access other module's store(s) (either to read or write from them).
- A reference to the application's **codec**. The `keeper` needs it to marshal structs before storing them, or to unmarshal them when it retrieves them, because stores only accept `[]bytes` as value.

Along with the type definition, the next important component of the `keeper.go` file is the `keeper`'s constructor function, `NewKeeper`. This function instantiates a new `keeper` of the type defined above with a `codec`, `stores` `keys` and potentially references other modules' `keeper`s as parameters. The `NewKeeper` function is called from the [application's constructor](#). The rest of the file defines the `keeper`'s methods, which are primarily getters and setters.

Command-Line, gRPC Services and REST Interfaces

Each module defines command-line commands, gRPC services, and REST routes to be exposed to the end-user via the [application's interfaces](#). This enables end-users to create messages of the types defined in the module, or to query the subset of the state managed by the module.

CLI

Generally, the [commands related to a module](#) are defined in a folder called `client/cli` in the module's folder. The CLI divides commands into two categories, transactions and queries, defined in

`client/cli/tx.go` and `client/cli/query.go`, respectively. Both commands are built on top of the [Cobra Library](#):

- Transactions commands let users generate new transactions so that they can be included in a block and eventually update the state. One command should be created for each [message type](#) defined in the module. The command calls the constructor of the message with the parameters provided by the end-user, and wraps it into a transaction. The Cosmos SDK handles signing and the addition of other transaction metadata.
- Queries let users query the subset of the state defined by the module. Query commands forward queries to the [application's query router](#), which routes them to the appropriate [querier](#) the `queryRoute` parameter supplied.

gRPC

[gRPC](#) is a modern open-source high performance RPC framework that has support in multiple languages. It is the recommended way for external clients (such as wallets, browsers and other backend services) to interact with a node.

Each module can expose gRPC endpoints called [service methods](#), which are defined in the [module's Protobuf query.proto file](#). A service method is defined by its name, input arguments, and output response. The module then needs to perform the following actions:

- Define a `RegisterGRPCGatewayRoutes` method on `AppModuleBasic` to wire the client gRPC requests to the correct handler inside the module.
- For each service method, define a corresponding handler. The handler implements the core logic necessary to serve the gRPC request, and is located in the `keeper/grpc_query.go` file.

gRPC-gateway REST Endpoints

Some external clients may not wish to use gRPC. In this case, the Cosmos SDK provides a gRPC gateway service, which exposes each gRPC service as a corresponding REST endpoint. Please refer to the [grpc-gateway](#) documentation to learn more.

The REST endpoints are defined in the Protobuf files, along with the gRPC services, using Protobuf annotations. Modules that want to expose REST queries should add `google.api.http` annotations to their `rpc` methods. By default, all REST endpoints defined in the SDK have a URL starting with the `/cosmos/` prefix.

The Cosmos SDK also provides a development endpoint to generate [Swagger](#) definition files for these REST endpoints. This endpoint can be enabled inside the `app.toml` config file, under the `api.swagger` key.

Application Interface

[Interfaces](#) let end-users interact with full-node clients. This means querying data from the full-node or creating and sending new transactions to be relayed by the full-node and eventually included in a block.

The main interface is the [Command-Line Interface](#). The CLI of a Cosmos SDK application is built by aggregating [CLI commands](#) defined in each of the modules used by the application. The CLI of an application is the same as the daemon (e.g. `appd`), and is defined in a file called `appd/main.go`. The file contains the following:

- A `main()` function, which is executed to build the `appd` interface client. This function prepares each command and adds them to the `rootCmd` before building them. At the root of `appd`, the

- function adds generic commands like `status`, `keys`, and `config`, query commands, tx commands, and `rest-server`.
- **Query commands**, which are added by calling the `queryCmd` function. This function returns a Cobra command that contains the query commands defined in each of the application's modules (passed as an array of `sdk.ModuleClients` from the `main()` function), as well as some other lower level query commands such as block or validator queries. Query command are called by using the command `appd query [query]` of the CLI.
 - **Transaction commands**, which are added by calling the `txCmd` function. Similar to `queryCmd`, the function returns a Cobra command that contains the tx commands defined in each of the application's modules, as well as lower level tx commands like transaction signing or broadcasting. Tx commands are called by using the command `appd tx [tx]` of the CLI.

See an example of an application's main command-line file from the [Cosmos Hub](#).

```
https://github.com/cosmos/gaia/blob/26ae7c2/cmd/gaiad/cmd/root.go#L39-L80
```

Dependencies and Makefile

This section is optional, as developers are free to choose their dependency manager and project building method. That said, the current most used framework for versioning control is [go.mod](#). It ensures each of the libraries used throughout the application are imported with the correct version.

The following is the `go.mod` of the [Cosmos Hub](#), provided as an example.

```
https://github.com/cosmos/gaia/blob/26ae7c2/go.mod#L1-L28
```

For building the application, a [Makefile](#) is generally used. The Makefile primarily ensures that the `go.mod` is run before building the two entrypoints to the application, `appd` and `appd`.

Here is an example of the [Cosmos Hub Makefile](#).

sidebar_position: 1

Transaction Lifecycle

:::note Synopsis This document describes the lifecycle of a transaction from creation to committed state changes. Transaction definition is described in a [different doc](#). The transaction is referred to as `Tx`. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK Application](#) :::

Creation

Transaction Creation

One of the main application interfaces is the command-line interface. The transaction `Tx` can be created by the user inputting a command in the following format from the [command-line](#), providing the type of transaction in `[command]`, arguments in `[args]`, and configurations such as gas prices in `[flags]`:

```
[appname] tx [command] [args] [flags]
```

This command automatically **creates** the transaction, **signs** it using the account's private key, and **broadcasts** it to the specified peer node.

There are several required and optional flags for transaction creation. The `--from` flag specifies which [account](#) the transaction is originating from. For example, if the transaction is sending coins, the funds are drawn from the specified `from` address.

Gas and Fees

Additionally, there are several [flags](#) users can use to indicate how much they are willing to pay in [fees](#):

- `--gas` refers to how much `gas`, which represents computational resources, `Tx` consumes. Gas is dependent on the transaction and is not precisely calculated until execution, but can be estimated by providing `auto` as the value for `--gas`.
- `--gas-adjustment` (optional) can be used to scale `gas` up in order to avoid underestimating. For example, users can specify their gas adjustment as 1.5 to use 1.5 times the estimated gas.
- `--gas-prices` specifies how much the user is willing to pay per unit of gas, which can be one or multiple denominations of tokens. For example, `--gas-prices=0.025uatom, 0.025upho` means the user is willing to pay 0.025uatom AND 0.025upho per unit of gas.
- `--fees` specifies how much in fees the user is willing to pay in total.
- `--timeout-height` specifies a block timeout height to prevent the tx from being committed past a certain height.

The ultimate value of the fees paid is equal to the gas multiplied by the gas prices. In other words, `fees = ceil(gas * gasPrices)`. Thus, since fees can be calculated using gas prices and vice versa, the users specify only one of the two.

Later, validators decide whether or not to include the transaction in their block by comparing the given or calculated `gas-prices` to their local `min-gas-prices`. `Tx` is rejected if its `gas-prices` is not high enough, so users are incentivized to pay more.

CLI Example

Users of the application `app` can enter the following command into their CLI to generate a transaction to send 1000uatom from a `senderAddress` to a `recipientAddress`. The command specifies how much gas they are willing to pay: an automatic estimate scaled up by 1.5 times, with a gas price of 0.025uatom per unit gas.

```
appd tx send <recipientAddress> 1000uatom --from <senderAddress> --gas auto --gas-adjustment 1.5 --gas-prices 0.025uatom
```

Other Transaction Creation Methods

The command-line is an easy way to interact with an application, but `Tx` can also be created using a [gRPC](#) or [REST interface](#) or some other entry point defined by the application developer. From the user's

perspective, the interaction depends on the web interface or wallet they are using (e.g. creating `Tx` using [Lunie.io](#) and signing it with a Ledger Nano S).

Addition to Mempool

Each full-node (running CometBFT) that receives a `Tx` sends an [ABCI message](#), `CheckTx`, to the application layer to check for validity, and receives an `abci.ResponseCheckTx`. If the `Tx` passes the checks, it is held in the node's [Mempool](#), an in-memory pool of transactions unique to each node, pending inclusion in a block - honest nodes discard a `Tx` if it is found to be invalid. Prior to consensus, nodes continuously check incoming transactions and gossip them to their peers.

Types of Checks

The full-nodes perform stateless, then stateful checks on `Tx` during `CheckTx`, with the goal to identify and reject an invalid transaction as early on as possible to avoid wasted computation.

Stateless checks do not require nodes to access state - light clients or offline nodes can do them - and are thus less computationally expensive. Stateless checks include making sure addresses are not empty, enforcing nonnegative numbers, and other logic specified in the definitions.

Stateful checks validate transactions and messages based on a committed state. Examples include checking that the relevant values exist and can be transacted with, the address has sufficient funds, and the sender is authorized or has the correct ownership to transact. At any given moment, full-nodes typically have [multiple versions](#) of the application's internal state for different purposes. For example, nodes execute state changes while in the process of verifying transactions, but still need a copy of the last committed state in order to answer queries - they should not respond using state with uncommitted changes.

In order to verify a `Tx`, full-nodes call `CheckTx`, which includes both *stateless* and *stateful* checks. Further validation happens later in the [DeliverTx](#) stage. `CheckTx` goes through several steps, beginning with decoding `Tx`.

Decoding

When `Tx` is received by the application from the underlying consensus engine (e.g. CometBFT), it is still in its [encoded](#) `[]byte` form and needs to be unmarshaled in order to be processed. Then, the [runTx](#) function is called to run in `runTxModeCheck` mode, meaning the function runs all checks but exits before executing messages and writing state changes.

ValidateBasic (deprecated)

Messages ([sdk.Msg](#)) are extracted from transactions (`Tx`). The `ValidateBasic` method of the `sdk.Msg` interface implemented by the module developer is run for each transaction. To discard obviously invalid messages, the `BaseApp` type calls the `ValidateBasic` method very early in the processing of the message in the [CheckTx](#) and [DeliverTx](#) transactions. `ValidateBasic` can include only **stateless** checks (the checks that do not require access to the state).

:::warning The `ValidateBasic` method on messages has been deprecated in favor of validating messages directly in their respective [Msg services](#).

Read [RFC 001](#) for more details. :::

:::note `BaseApp` still calls `ValidateBasic` on messages that implements that method for backwards compatibility. :::

Guideline

`ValidateBasic` should not be used anymore. Message validation should be performed in the `Msg` service when [handling a message](#) in a module Msg Server.

AnteHandler

`AnteHandler`s even though optional, are in practice very often used to perform signature verification, gas calculation, fee deduction, and other core operations related to blockchain transactions.

A copy of the cached context is provided to the `AnteHandler`, which performs limited checks specified for the transaction type. Using a copy allows the `AnteHandler` to do stateful checks for `Tx` without modifying the last committed state, and revert back to the original if the execution fails.

For example, the `auth` module `AnteHandler` checks and increments sequence numbers, checks signatures and account numbers, and deducts fees from the first signer of the transaction - all state changes are made using the `checkState`.

Gas

The `Context`, which keeps a `GasMeter` that tracks how much gas is used during the execution of `Tx`, is initialized. The user-provided amount of gas for `Tx` is known as `GasWanted`. If `GasConsumed`, the amount of gas consumed during execution, ever exceeds `GasWanted`, the execution stops and the changes made to the cached copy of the state are not committed. Otherwise, `CheckTx` sets `GasUsed` equal to `GasConsumed` and returns it in the result. After calculating the gas and fee values, validator-nodes check that the user-specified `gas-prices` is greater than their locally defined `min-gas-prices`.

Discard or Addition to Mempool

If at any point during `CheckTx` the `Tx` fails, it is discarded and the transaction lifecycle ends there. Otherwise, if it passes `CheckTx` successfully, the default protocol is to relay it to peer nodes and add it to the Mempool so that the `Tx` becomes a candidate to be included in the next block.

The **mempool** serves the purpose of keeping track of transactions seen by all full-nodes. Full-nodes keep a **mempool cache** of the last `mempool.cache_size` transactions they have seen, as a first line of defense to prevent replay attacks. Ideally, `mempool.cache_size` is large enough to encompass all of the transactions in the full mempool. If the mempool cache is too small to keep track of all the transactions, `CheckTx` is responsible for identifying and rejecting replayed transactions.

Currently existing preventative measures include fees and a `sequence` (nonce) counter to distinguish replayed transactions from identical but valid ones. If an attacker tries to spam nodes with many copies of a `Tx`, full-nodes keeping a mempool cache reject all identical copies instead of running `CheckTx` on them. Even if the copies have incremented `sequence` numbers, attackers are disincentivized by the need to pay fees.

Validator nodes keep a mempool to prevent replay attacks, just as full-nodes do, but also use it as a pool of unconfirmed transactions in preparation of block inclusion. Note that even if a `Tx` passes all checks at this stage, it is still possible to be found invalid later on, because `CheckTx` does not fully validate the transaction (that is, it does not actually execute the messages).

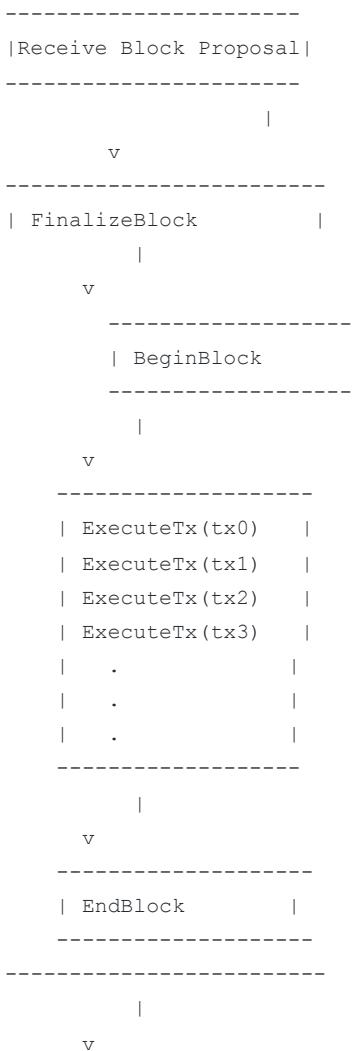
Inclusion in a Block

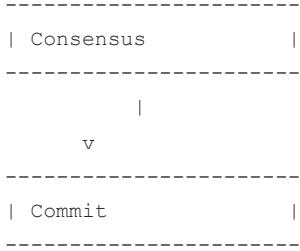
Consensus, the process through which validator nodes come to agreement on which transactions to accept, happens in **rounds**. Each round begins with a proposer creating a block of the most recent transactions and ends with **validators**, special full-nodes with voting power responsible for consensus, agreeing to accept the block or go with a `nil` block instead. Validator nodes execute the consensus algorithm, such as [CometBFT](#), confirming the transactions using ABCI requests to the application, in order to come to this agreement.

The first step of consensus is the **block proposal**. One proposer amongst the validators is chosen by the consensus algorithm to create and propose a block - in order for a `Tx` to be included, it must be in this proposer's mempool.

State Changes

The next step of consensus is to execute the transactions to fully validate them. All full-nodes that receive a block proposal from the correct proposer execute the transactions by calling the ABCI function `FinalizeBlock`. As mentioned throughout the documentation `BeginBlock`, `ExecuteTx` and `EndBlock` are called within `FinalizeBlock`. Although every full-node operates individually and locally, the outcome is always consistent and unequivocal. This is because the state changes brought about by the messages are predictable, and the transactions are specifically sequenced in the proposed block.





Transaction Execution

The `FinalizeBlock` ABCI function defined in `BaseApp` does the bulk of the state transitions: it is run for each transaction in the block in sequential order as committed to during consensus. Under the hood, transaction execution is almost identical to `CheckTx` but calls the `runTx` function in deliver mode instead of check mode. Instead of using their `checkState`, full-nodes use `finalizeblock`:

- **Decoding:** Since `FinalizeBlock` is an ABCI call, `Tx` is received in the encoded `[]byte` form. Nodes first unmarshal the transaction, using the `TxConfig` defined in the app, then call `runTx` in `execModeFinalize`, which is very similar to `CheckTx` but also executes and writes state changes.
- **Checks and AnteHandler :** Full-nodes call `validateBasicMsgs` and `AnteHandler` again. This second check happens because they may not have seen the same transactions during the addition to Mempool stage and a malicious proposer may have included invalid ones. One difference here is that the `AnteHandler` does not compare `gas-prices` to the node's `min-gas-prices` since that value is local to each node - differing values across nodes yield nondeterministic results.
- **MsgServiceRouter :** After `CheckTx` exits, `FinalizeBlock` continues to run `runMsgs` to fully execute each `Msg` within the transaction. Since the transaction may have messages from different modules, `BaseApp` needs to know which module to find the appropriate handler. This is achieved using `BaseApp`'s `MsgServiceRouter` so that it can be processed by the module's Protobuf `Msg service`. For `LegacyMsg` routing, the `Route` function is called via the `module manager` to retrieve the route name and find the legacy `Handler` within the module.
- **Msg service:** Protobuf `Msg` service is responsible for executing each message in the `Tx` and causes state transitions to persist in `finalizeBlockState`.
- **PostHandlers:** `PostHandler`s are run after the execution of the message. If they fail, the state change of `runMsgs`, as well as `PostHandlers`, are both reverted.
- **Gas:** While a `Tx` is being delivered, a `GasMeter` is used to keep track of how much gas is being used; if execution completes, `GasUsed` is set and returned in the `abci.ExecTxResult`. If execution halts because `BlockGasMeter` or `GasMeter` has run out or something else goes wrong, a deferred function at the end appropriately errors or panics.

If there are any failed state changes resulting from a `Tx` being invalid or `GasMeter` running out, the transaction processing terminates and any state changes are reverted. Invalid transactions in a block proposal cause validator nodes to reject the block and vote for a `nil` block instead.

Commit

The final step is for nodes to commit the block and state changes. Validator nodes perform the previous step of executing state transitions in order to validate the transactions, then sign the block to confirm it. Full nodes that are not validators do not participate in consensus - i.e. they cannot vote - but listen for votes to understand whether or not they should commit the state changes.

When they receive enough validator votes (2/3+ *precommits* weighted by voting power), full nodes commit to a new block to be added to the blockchain and finalize the state transitions in the application layer. A new state root is generated to serve as a merkle proof for the state transitions. Applications use the [Commit](#) ABCI method inherited from [Baseapp](#); it syncs all the state transitions by writing the `deliverState` into the application's internal state. As soon as the state changes are committed, `checkState` starts afresh from the most recently committed state and `deliverState` resets to `nil` in order to be consistent and reflect the changes.

Note that not all blocks have the same number of transactions and it is possible for consensus to result in a `nil` block or one with none at all. In a public blockchain network, it is also possible for validators to be **byzantine**, or malicious, which may prevent a `Tx` from being committed in the blockchain. Possible malicious behaviors include the proposer deciding to censor a `Tx` by excluding it from the block or a validator voting against the block.

At this point, the transaction lifecycle of a `Tx` is over: nodes have verified its validity, delivered it by executing its state changes, and committed those changes. The `Tx` itself, in `[]byte` form, is stored in a block and appended to the blockchain.

sidebar_position: 1

Query Lifecycle

:::note Synopsis This document describes the lifecycle of a query in a Cosmos SDK application, from the user interface to application stores and back. The query is referred to as `MyQuery` . :::

:::note

Pre-requisite Readings

- [Transaction Lifecycle](#) :::

Query Creation

A [query](#) is a request for information made by end-users of applications through an interface and processed by a full-node. Users can query information about the network, the application itself, and application state directly from the application's stores or modules. Note that queries are different from [transactions](#) (view the lifecycle [here](#)), particularly in that they do not require consensus to be processed (as they do not trigger state-transitions); they can be fully handled by one full-node.

For the purpose of explaining the query lifecycle, let's say the query, `MyQuery` , is requesting a list of delegations made by a certain delegator address in the application called `simapp` . As is to be expected, the [staking](#) module handles this query. But first, there are a few ways `MyQuery` can be created by users.

CLI

The main interface for an application is the command-line interface. Users connect to a full-node and run the CLI directly from their machines - the CLI interacts directly with the full-node. To create `MyQuery` from their terminal, users type the following command:

```
simd query staking delegations <delegatorAddress>
```

This query command was defined by the `staking` module developer and added to the list of subcommands by the application developer when creating the CLI.

Note that the general format is as follows:

```
simd query [moduleName] [command] <arguments> --flag <flagArg>
```

To provide values such as `--node` (the full-node the CLI connects to), the user can use the `app.toml` config file to set them or provide them as flags.

The CLI understands a specific set of commands, defined in a hierarchical structure by the application developer: from the `root command` (`simd`), the type of command (`Myquery`), the module that contains the command (`staking`), and command itself (`delegations`). Thus, the CLI knows exactly which module handles this command and directly passes the call there.

gRPC

Another interface through which users can make queries is [gRPC](#) requests to a [gRPC server](#). The endpoints are defined as [Protocol Buffers](#) service methods inside `.proto` files, written in Protobuf's own language-agnostic interface definition language (IDL). The Protobuf ecosystem developed tools for code-generation from `*.proto` files into various languages. These tools allow to build gRPC clients easily.

One such tool is [grpcurl](#), and a gRPC request for `MyQuery` using this client looks like:

```
grpcurl \
    -plaintext                                         # We want results in plain
test
    -import-path ./proto \                            # Import these .proto files
    -proto ./proto/cosmos/staking/v1beta1/query.proto \ # Look into this .proto
file for the Query protobuf service
    -d '{"address":"$MY_DELEGATOR"}' \                # Query arguments
    localhost:9090 \                                    # gRPC server endpoint
    cosmos.staking.v1beta1.Query/Delegations          # Fully-qualified service
method name
```

REST

Another interface through which users can make queries is through HTTP Requests to a [REST server](#). The REST server is fully auto-generated from Protobuf services, using [gRPC-gateway](#).

An example HTTP request for `MyQuery` looks like:

```
GET  
http://localhost:1317/cosmos/staking/v1beta1/delegators/{delegatorAddr}/delegations
```

How Queries are Handled by the CLI

The preceding examples show how an external user can interact with a node by querying its state. To understand in more detail the exact lifecycle of a query, let's dig into how the CLI prepares the query, and how the node handles it. The interactions from the users' perspective are a bit different, but the underlying functions are almost identical because they are implementations of the same command defined by the module developer. This step of processing happens within the CLI, gRPC, or REST server, and heavily involves a `client.Context`.

Context

The first thing that is created in the execution of a CLI command is a `client.Context`. A `client.Context` is an object that stores all the data needed to process a request on the user side. In particular, a `client.Context` stores the following:

- **Codec:** The [encoder/decoder](#) used by the application, used to marshal the parameters and query before making the CometBFT RPC request and unmarshal the returned response into a JSON object. The default codec used by the CLI is Protobuf.
- **Account Decoder:** The account decoder from the [auth](#) module, which translates `[]byte`s into accounts.
- **RPC Client:** The CometBFT RPC Client, or node, to which requests are relayed.
- **Keyring:** A [Key Manager](#) used to sign transactions and handle other operations with keys.
- **Output Writer:** A [Writer](#) used to output the response.
- **Configurations:** The flags configured by the user for this command, including `--height`, specifying the height of the blockchain to query, and `--indent`, which indicates to add an indent to the JSON response.

The `client.Context` also contains various functions such as `Query()`, which retrieves the RPC Client and makes an ABCI call to relay a query to a full-node.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/context.go#L25-L68
```

The `client.Context`'s primary role is to store data used during interactions with the end-user and provide methods to interact with this data - it is used before and after the query is processed by the full-node. Specifically, in handling `MyQuery`, the `client.Context` is utilized to encode the query parameters, retrieve the full-node, and write the output. Prior to being relayed to a full-node, the query needs to be encoded into a `[]byte` form, as full-nodes are application-agnostic and do not understand specific types. The full-node (RPC Client) itself is retrieved using the `client.Context`, which knows which node the user CLI is connected to. The query is relayed to this full-node to be processed. Finally, the `client.Context` contains a `Writer` to write output when the response is returned. These steps are further described in later sections.

Arguments and Route Creation

At this point in the lifecycle, the user has created a CLI command with all of the data they wish to include in their query. A `client.Context` exists to assist in the rest of the `MyQuery`'s journey. Now, the next step

is to parse the command or request, extract the arguments, and encode everything. These steps all happen on the user side within the interface they are interacting with.

Encoding

In our case (querying an address's delegations), `MyQuery` contains an `address delegatorAddress` as its only argument. However, the request can only contain `[]byte`s, as it is ultimately relayed to a consensus engine (e.g. CometBFT) of a full-node that has no inherent knowledge of the application types. Thus, the `codec` of `client.Context` is used to marshal the address.

Here is what the code looks like for the CLI command:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/staking/client/cli/query.go#L315-L318
```

gRPC Query Client Creation

The Cosmos SDK leverages code generated from Protobuf services to make queries. The `staking` module's `MyQuery` service generates a `queryClient`, which the CLI uses to make queries. Here is the relevant code:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/staking/client/cli/query.go#L308-L343
```

Under the hood, the `client.Context` has a `Query()` function used to retrieve the pre-configured node and relay a query to it; the function takes the query fully-qualified service method name as path (in our case: `/cosmos.staking.v1beta1.Query/Delegations`), and arguments as parameters. It first retrieves the RPC Client (called the `node`) configured by the user to relay this query to, and creates the `ABCIQueryOptions` (parameters formatted for the ABCI call). The node is then used to make the ABCI call, `ABCIQueryWithOptions()`.

Here is what the code looks like:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/query.go#L79-L113
```

RPC

With a call to `ABCIQueryWithOptions()`, `MyQuery` is received by a [full-node](#) which then processes the request. Note that, while the RPC is made to the consensus engine (e.g. CometBFT) of a full-node, queries are not part of consensus and so are not broadcasted to the rest of the network, as they do not require anything the network needs to agree upon.

Read more about ABCI Clients and CometBFT RPC in the [CometBFT documentation](#).

Application Query Handling

When a query is received by the full-node after it has been relayed from the underlying consensus engine, it is at that point being handled within an environment that understands application-specific types and has a copy of the state. `baseapp` implements the ABCI `Query()` function and handles gRPC queries. The

query route is parsed, and it matches the fully-qualified service method name of an existing service method (most likely in one of the modules), then `baseapp` relays the request to the relevant module.

Since `MyQuery` has a Protobuf fully-qualified service method name from the `staking` module (recall `/cosmos.staking.v1beta1.Query/Delegations`), `baseapp` first parses the path, then uses its own internal `GRPCQueryRouter` to retrieve the corresponding gRPC handler, and routes the query to the module. The gRPC handler is responsible for recognizing this query, retrieving the appropriate values from the application's stores, and returning a response. Read more about query services [here](#).

Once a result is received from the querier, `baseapp` begins the process of returning a response to the user.

Response

Since `Query()` is an ABCI function, `baseapp` returns the response as an `abci.ResponseQuery` type. The `client.Context Query()` routine receives the response and.

CLI Response

The application `codec` is used to unmarshal the response to a JSON and the `client.Context` prints the output to the command line, applying any configurations such as the output type (text, JSON or YAML).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/context.go#L341-L349
```

And that's a wrap! The result of the query is outputted to the console by the CLI.

sidebar_position: 1

Accounts

:::note Synopsis This document describes the in-built account and public key system of the Cosmos SDK. :::

:::note

Pre-requisite Readings

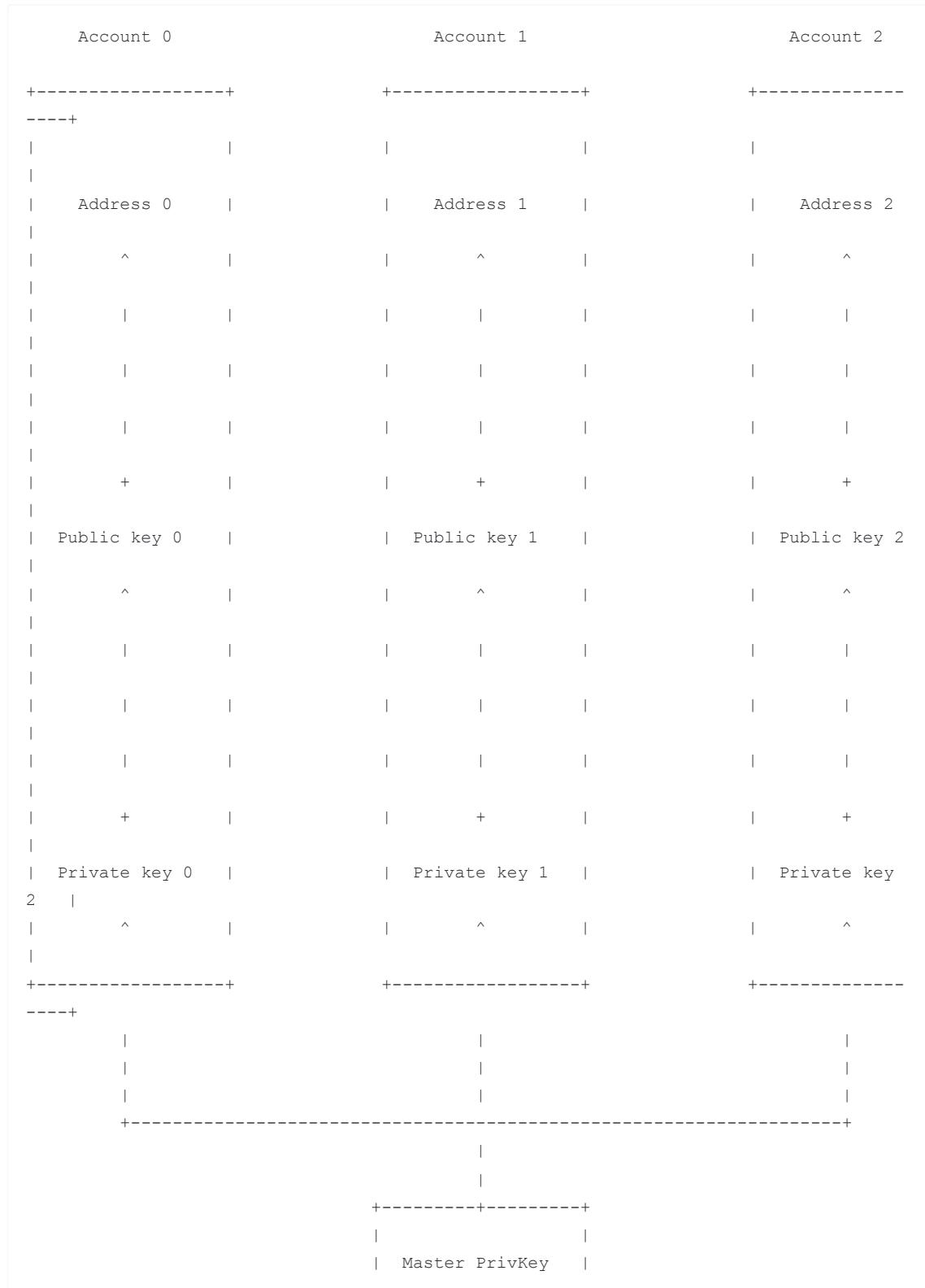
- [Anatomy of a Cosmos SDK Application](#)

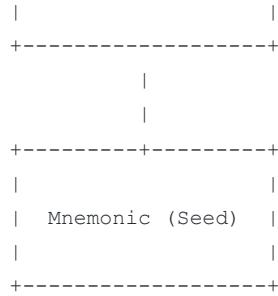
:::

Account Definition

In the Cosmos SDK, an `account` designates a pair of `public key` `PubKey` and `private key` `PrivKey`. The `PubKey` can be derived to generate various `Addresses`, which are used to identify users (among other parties) in the application. `Addresses` are also associated with `message s` to identify the sender of the message. The `PrivKey` is used to generate `digital signatures` to prove that an `Address` associated with the `PrivKey` approved of a given `message`.

For HD key derivation the Cosmos SDK uses a standard called [BIP32](#). The BIP32 allows users to create an HD wallet (as specified in [BIP44](#)) - a set of accounts derived from an initial secret seed. A seed is usually created from a 12- or 24-word mnemonic. A single seed can derive any number of `PrivKey`s using a one-way cryptographic function. Then, a `PubKey` can be derived from the `PrivKey`. Naturally, the mnemonic is the most sensitive information, as private keys can always be re-generated if the mnemonic is preserved.





In the Cosmos SDK, keys are stored and managed by using an object called a [Keyring](#).

Keys, accounts, addresses, and signatures

The principal way of authenticating a user is done using [digital signatures](#). Users sign transactions using their own private key. Signature verification is done with the associated public key. For on-chain signature verification purposes, we store the public key in an `Account` object (alongside other data required for a proper transaction validation).

In the node, all data is stored using Protocol Buffers serialization.

The Cosmos SDK supports the following digital key schemes for creating digital signatures:

- `secp256k1`, as implemented in the [Cosmos SDK's crypto/keys/secp256k1 package](#),
- `secp256r1`, as implemented in the [Cosmos SDK's crypto/keys/secp256r1 package](#),
- `tm-ed25519`, as implemented in the [Cosmos SDK crypto/keys/ed25519 package](#). This scheme is supported only for the consensus validation.

	Address length in bytes	Public key length in bytes	Used for transaction authentication	Used for consensus (cometbft)
<code>secp256k1</code>	20	33	yes	no
<code>secp256r1</code>	32	33	yes	no
<code>tm-ed25519</code>	-- not used --	32	no	yes

Addresses

`Addresses` and `PubKey`s are both public information that identifies actors in the application. `Account` is used to store authentication information. The basic account implementation is provided by a `BaseAccount` object.

Each account is identified using `Address` which is a sequence of bytes derived from a public key. In the Cosmos SDK, we define 3 types of addresses that specify a context where an account is used:

- `AccAddress` identifies users (the sender of a `message`).
- `ValAddress` identifies validator operators.
- `ConsAddress` identifies validator nodes that are participating in consensus. Validator nodes are derived using the `ed25519` curve.

These types implement the `Address` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/address.go#L126-L134
```

Address construction algorithm is defined in [ADR-28](#). Here is the standard way to obtain an account address from a `pub` public key:

```
sdk.AccAddress(pub.Address().Bytes())
```

Of note, the `Marshal()` and `Bytes()` method both return the same raw `[]byte` form of the address. `Marshal()` is required for Protobuf compatibility.

For user interaction, addresses are formatted using [Bech32](#) and implemented by the `String` method. The Bech32 method is the only supported format to use when interacting with a blockchain. The Bech32 human-readable part (Bech32 prefix) is used to denote an address type. Example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/address.go#L299-L316
```

	Address Bech32 Prefix
Accounts	cosmos
Validator Operator	cosmosvaloper
Consensus Nodes	cosmosvalcons

Public Keys

Public keys in Cosmos SDK are defined by `cryptotypes.PubKey` interface. Since public keys are saved in a store, `cryptotypes.PubKey` extends the `proto.Message` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/crypto/types/types.go#L8-L17
```

A compressed format is used for `secp256k1` and `secp256r1` serialization.

- The first byte is a `0x02` byte if the `y`-coordinate is the lexicographically largest of the two associated with the `x`-coordinate.
- Otherwise the first byte is a `0x03`.

This prefix is followed by the `x`-coordinate.

Public Keys are not used to reference accounts (or users) and in general are not used when composing transaction messages (with few exceptions: `MsgCreateValidator`, `Validator` and `Multisig` messages). For user interactions, `PubKey` is formatted using Protobufs JSON ([ProtoMarshalJSON](#) function). Example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/keys/output.go#L23-L39
```

Keyring

A `Keyring` is an object that stores and manages accounts. In the Cosmos SDK, a `Keyring` implementation follows the `Keyring` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/crypto/keyring/keyring.go#L57-L105
```

The default implementation of `Keyring` comes from the third-party [99designs/keyring](#) library.

A few notes on the `Keyring` methods:

- `Sign(uid string, msg []byte) ([]byte, types.PubKey, error)` strictly deals with the signature of the `msg` bytes. You must prepare and encode the transaction into a canonical `[]byte` form. Because protobuf is not deterministic, it has been decided in [ADR-020](#) that the canonical payload to sign is the `SignDoc` struct, deterministically encoded using [ADR-027](#). Note that signature verification is not implemented in the Cosmos SDK by default, it is deferred to the [anteHandler](#).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L50-L66
```

- `NewAccount(uid, mnemonic, bip39Passphrase, hdPath string, algo SignatureAlgo) (*Record, error)` creates a new account based on the [bip44_path](#) and persists it on disk. The `PrivKey` is **never stored unencrypted**, instead it is [encrypted with a passphrase](#) before being persisted. In the context of this method, the key type and sequence number refer to the segment of the BIP44 derivation path (for example, `0, 1, 2, ...`) that is used to derive a private and a public key from the mnemonic. Using the same mnemonic and derivation path, the same `PrivKey`, `PubKey` and `Address` is generated. The following keys are supported by the keyring:
 - `secp256k1`
 - `ed25519`
 - `ExportPrivKeyArmor(uid, encryptPassphrase string) (armor string, err error)` exports a private key in ASCII-armored encrypted format using the given passphrase. You can then either import the private key again into the keyring using the `ImportPrivKey(uid, armor, passphrase string)` function or decrypt it into a raw private key using the `UnarmorDecryptPrivKey(armorStr string, passphrase string)` function.

Create New Key Type

To create a new key type for using in keyring, `keyring.SignatureAlgo` interface must be fulfilled.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/crypto/keyring/signing_algorithms.go#L10-L15
```

The interface consists in three methods where `Name()` returns the name of the algorithm as a `hd.PubKeyType` and `Derive()` and `Generate()` must return the following functions respectively:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/crypto/hd/algo.go#L28-L31
```

Once the `keyring.SignatureAlgo` has been implemented it must be added to the [list of supported algos](#) of the keyring.

For simplicity the implementation of a new key type should be done inside the `crypto/hd` package. There is an example of a working `secp256k1` implementation in [algo.go](#).

Implementing secp256r1 algo

Here is an example of how secp256r1 could be implemented.

First a new function to create a private key from a secret number is needed in the `secp256r1` package. This function could look like this:

```
// cosmos-sdk/crypto/keys/secp256r1/privkey.go

// NewPrivKeyFromSecret creates a private key derived for the secret number
// represented in big-endian. The `secret` must be a valid ECDSA field element.
func NewPrivKeyFromSecret(secret []byte) (*PrivKey, error) {
    var d = new(big.Int).SetBytes(secret)
    if d.Cmp(secp256r1.Params().N) >= 1 {
        return nil, errorsmod.Wrap(errors.ErrInvalidRequest, "secret not in the
curve base field")
    }
    sk := new(ecdsa.PrivKey)
    return &PrivKey{&ecdsaSK{*sk}}, nil
}
```

After that `secp256r1Algo` can be implemented.

```
// cosmos-sdk/crypto/hd/secp256r1Algo.go

package hd

import (
    "github.com/cosmos/go-bip39"

    "github.com/cosmos/cosmos-sdk/crypto/keys/secp256r1"
    "github.com/cosmos/cosmos-sdk/crypto/types"
)

// Secp256r1Type uses the secp256r1 ECDSA parameters.
const Secp256r1Type = PubKeyType("secp256r1")

var Secp256r1 = secp256r1Algo{}

type secp256r1Algo struct{ }

func (s secp256r1Algo) Name() PubKeyType {
    return Secp256r1Type
}
```

```

}

// Derive derives and returns the secp256r1 private key for the given seed and HD
path.
func (s secp256r1Algo) Derive() DeriveFn {
    return func(mnemonic string, bip39Passphrase, hdPath string) ([]byte, error) {
        seed, err := bip39.NewSeedWithErrorChecking(mnemonic, bip39Passphrase)
        if err != nil {
            return nil, err
        }

        masterPriv, ch := ComputeMastersFromSeed(seed)
        if len(hdPath) == 0 {
            return masterPriv[:], nil
        }
        derivedKey, err := DerivePrivateKeyForPath(masterPriv, ch, hdPath)

        return derivedKey, err
    }
}

// Generate generates a secp256r1 private key from the given bytes.
func (s secp256r1Algo) Generate() GenerateFn {
    return func(bz []byte) types.PrivKey {
        key, err := secp256r1.NewPrivKeyFromSecret(bz)
        if err != nil {
            panic(err)
        }
        return key
    }
}

```

Finally, the algo must be added to the list of [supported algos](#) by the keyring.

```

// cosmos-sdk/crypto/keyring/keyring.go

func newKeystore(kr keyring.Keyring, cdc codec.Codec, backend string, opts
...Option) keystore {
    // Default options for keybase, these can be overwritten using the
    // Option function
    options := Options{
        SupportedAlgos:      SigningAlgoList{hd.Secp256k1, hd.Secp256r1}, // added
here
        SupportedAlgosLedger: SigningAlgoList{hd.Secp256k1},
    }
...

```

Hereafter to create new keys using your algo, you must specify it with the flag `--algo` :

```
simd keys add myKey --algo secp256r1
```

sidebar_position: 1

Gas and Fees

:::note Synopsis This document describes the default strategies to handle gas and fees within a Cosmos SDK application. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK Application](#)

:::

Introduction to Gas and Fees

In the Cosmos SDK, `gas` is a special unit that is used to track the consumption of resources during execution. `gas` is typically consumed whenever read and writes are made to the store, but it can also be consumed if expensive computation needs to be done. It serves two main purposes:

- Make sure blocks are not consuming too many resources and are finalized. This is implemented by default in the Cosmos SDK via the [block_gas_meter](#).
- Prevent spam and abuse from end-user. To this end, `gas` consumed during [message](#) execution is typically priced, resulting in a `fee` (`fees = gas * gas-prices`). `fees` generally have to be paid by the sender of the [message](#). Note that the Cosmos SDK does not enforce `gas` pricing by default, as there may be other ways to prevent spam (e.g. bandwidth schemes). Still, most applications implement `fee` mechanisms to prevent spam by using the [AnteHandler](#) .

Gas Meter

In the Cosmos SDK, `gas` is a simple alias for `uint64`, and is managed by an object called a *gas meter*. Gas meters implement the `GasMeter` interface

<https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/gas.go#L40-L51>

where:

- `GasConsumed()` returns the amount of gas that was consumed by the gas meter instance.
- `GasConsumedToLimit()` returns the amount of gas that was consumed by gas meter instance, or the limit if it is reached.
- `GasRemaining()` returns the gas left in the `GasMeter`.
- `Limit()` returns the limit of the gas meter instance. `0` if the gas meter is infinite.
- `ConsumeGas(amount Gas, descriptor string)` consumes the amount of `gas` provided. If the `gas` overflows, it panics with the `descriptor` message. If the gas meter is not infinite, it panics if `gas` consumed goes above the limit.
- `RefundGas()` deducts the given amount from the gas consumed. This functionality enables refunding gas to the transaction or block gas pools so that EVM-compatible chains can fully support the go-ethereum StateDB interface.

- `IsPastLimit()` returns `true` if the amount of gas consumed by the gas meter instance is strictly above the limit, `false` otherwise.
- `IsOutOfGas()` returns `true` if the amount of gas consumed by the gas meter instance is above or equal to the limit, `false` otherwise.

The gas meter is generally held in `ctx`, and consuming gas is done with the following pattern:

```
ctx.GasMeter().ConsumeGas(amount, "description")
```

By default, the Cosmos SDK makes use of two different gas meters, the [main gas meter](#) and the [block gas meter](#).

Main Gas Meter

`ctx.GasMeter()` is the main gas meter of the application. The main gas meter is initialized in `FinalizeBlock` via `setFinalizeBlockState`, and then tracks gas consumption during execution sequences that lead to state-transitions, i.e. those originally triggered by `FinalizeBlock`. At the beginning of each transaction execution, the main gas meter **must be set to 0** in the [AnteHandler](#), so that it can track gas consumption per-transaction.

Gas consumption can be done manually, generally by the module developer in the [BeginBlocker](#), [EndBlocker](#) or [Msg service](#), but most of the time it is done automatically whenever there is a read or write to the store. This automatic gas consumption logic is implemented in a special store called [GasKV](#).

Block Gas Meter

`ctx.BlockGasMeter()` is the gas meter used to track gas consumption per block and make sure it does not go above a certain limit. A new instance of the `BlockGasMeter` is created each time `FinalizeBlock` is called. The `BlockGasMeter` is finite, and the limit of gas per block is defined in the application's consensus parameters. By default, Cosmos SDK applications use the default consensus parameters provided by CometBFT:

```
https://github.com/cometbft/cometbft/blob/v0.37.0/types/params.go#L66-L105
```

When a new [transaction](#) is being processed via `FinalizeBlock`, the current value of `BlockGasMeter` is checked to see if it is above the limit. If it is, the transaction fails and returned to the consensus engine as a failed transaction. This can happen even with the first transaction in a block, as `FinalizeBlock` itself can consume gas. If not, the transaction is processed normally. At the end of `FinalizeBlock`, the gas tracked by `ctx.BlockGasMeter()` is increased by the amount consumed to process the transaction:

```
ctx.BlockGasMeter().ConsumeGas(
    ctx.GasMeter().GasConsumedToLimit(),
    "block gas meter",
)
```

AnteHandler

The `AnteHandler` is run for every transaction during `CheckTx` and `FinalizeBlock`, before a Protobuf `Msg` service method for each `sdk.Msg` in the transaction.

The anteHandler is not implemented in the core Cosmos SDK but in a module. That said, most applications today use the default implementation defined in the [auth module](#). Here is what the `anteHandler` is intended to do in a normal Cosmos SDK application:

- Verify that the transactions are of the correct type. Transaction types are defined in the module that implements the `anteHandler`, and they follow the transaction interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/tx_msg.go#L51-L56
```

This enables developers to play with various types for the transaction of their application. In the default `auth` module, the default transaction type is `Tx`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L14-L27
```

- Verify signatures for each `message` contained in the transaction. Each `message` should be signed by one or multiple sender(s), and these signatures must be verified in the `anteHandler`.
- During `CheckTx`, verify that the gas prices provided with the transaction is greater than the local `min-gas-prices` (as a reminder, gas-prices can be deducted from the following equation: `fees = gas * gas-prices`). `min-gas-prices` is a parameter local to each full-node and used during `CheckTx` to discard transactions that do not provide a minimum amount of fees. This ensures that the mempool cannot be spammed with garbage transactions.
- Verify that the sender of the transaction has enough funds to cover for the `fees`. When the end-user generates a transaction, they must indicate 2 of the 3 following parameters (the third one being implicit): `fees`, `gas` and `gas-prices`. This signals how much they are willing to pay for nodes to execute their transaction. The provided `gas` value is stored in a parameter called `GasWanted` for later use.
- Set `newCtx.GasMeter` to 0, with a limit of `GasWanted`. **This step is crucial**, as it not only makes sure the transaction cannot consume infinite gas, but also that `ctx.GasMeter` is reset in-between each transaction (`ctx` is set to `newCtx` after `anteHandler` is run, and the `anteHandler` is run each time a transaction executes).

As explained above, the `anteHandler` returns a maximum limit of `gas` the transaction can consume during execution called `GasWanted`. The actual amount consumed in the end is denominated `GasUsed`, and we must therefore have `GasUsed <= GasWanted`. Both `GasWanted` and `GasUsed` are relayed to the underlying consensus engine when [FinalizeBlock](#) returns.

sidebar_position: 1

Overview of `app.go`

This section is intended to provide an overview of the `SimApp app.go` file and is still a work in progress. For now please instead read the [tutorials](#) for a deep dive on how to build a chain.

Complete `app.go`

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go
```

sidebar_position: 1

Overview of `app_v2.go`

:::note Synopsis

The Cosmos SDK allows much easier wiring of an `app.go` thanks to App Wiring and [depinject](#). Learn more about the rationale of App Wiring in [ADR-057](#).

:::

:::note

Pre-requisite Readings

- [ADR 057: App Wiring](#)
- [Depinjection Documentation](#)
- [Modules depinjection-ready](#)

:::

This section is intended to provide an overview of the `SimApp app_v2.go` file with App Wiring.

`app_config.go`

The `app_config.go` file is the single place to configure all modules parameters.

1. Create the `AppConfig` variable:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/app_config.go#L103
```

2. Configure the `runtime` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/app_config.go#L103-L167
```

3. Configure the modules defined in the `BeginBlocker` and `EndBlocker` and the `tx` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/app_config.go#L112-L129
```

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app\_config.go#L200-L203
```

Complete `app_config.go`

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app\_config.go
```

Alternative formats

:::tip The example above shows how to create an `AppConfig` using Go. However, it is also possible to create an `AppConfig` using YAML, or JSON.

The configuration can then be embed with `go:embed` and read with [`appconfig.LoadYAML`](#), or [`appconfig.LoadJSON`](#), in `app_v2.go`.

```
//go:embed app_config.yaml
var (
    AppConfigYaml []byte
    AppConfig = appconfig.LoadYAML(AppConfigYaml)
)
```

:::

```
modules:
- name: runtime
  config:
    "@type": cosmos.app.runtime.v1alpha1.Module
    app_name: SimApp
    begin_blockers: [staking, auth, bank]
    end_blockers: [bank, auth, staking]
    init_genesis: [bank, auth, staking]
- name: auth
  config:
    "@type": cosmos.auth.module.v1.Module
    bech32_prefix: cosmos
- name: bank
  config:
    "@type": cosmos.bank.module.v1.Module
- name: staking
  config:
    "@type": cosmos.staking.module.v1.Module
- name: tx
  config:
    "@type": cosmos.tx.module.v1.Module
```

A more complete example of `app.yaml` can be found [here](#).

`app_v2.go`

```
app_v2.go is the place where SimApp is constructed. depinjected.Inject facilitates that by automatically wiring the app modules and keepers, provided an application configuration AppConfig is provided. SimApp is constructed, when calling the injected *runtime.AppBuilder , with appBuilder.Build(...).
```

In short depinjected and the [runtime package](#) abstract the wiring of the app, and the AppBuilder is the place where the app is constructed. runtime takes care of registering the codecs, KV store, subspaces and instantiating baseapp .

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app\_v2.go#L101-L245
```

:::warning When using depinjected.Inject , the injected types must be pointers. :::

Advanced Configuration

In advanced cases, it is possible to inject extra (module) configuration in a way that is not (yet) supported by AppConfig .

In this case, use depinjected.Configs for combining the extra configuration and AppConfig , and depinjected.Supply to providing that extra configuration. More information on how work depinjected.Configs and depinjected.Supply can be found in the [depinject documentation](#).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app\_v2.go#L114-L146
```

Complete app_v2.go

:::tip Note that in the complete SimApp app_v2.go file, testing utilities are also defined, but they could as well be defined in a separate file. :::

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app\_v2.go
```

sidebar_position: 1

Application Mempool

:::note Synopsis This sections describes how the app-side mempool can be used and replaced. :::

Since v0.47 the application has its own mempool to allow much more granular block building than previous versions. This change was enabled by [ABCI 1.0](#). Notably it introduces the PrepareProposal and ProcessProposal steps of ABCI++.

:::note

Pre-requisite Readings

- [BaseApp](#)

:::

Prepare Proposal

```
PrepareProposal handles construction of the block, meaning that when a proposer is preparing to propose a block, it requests the application to evaluate a RequestPrepareProposal , which contains a series of transactions from CometBFT's mempool. At this point, the application has complete control over the proposal. It can modify, delete, and inject transactions from its own app-side mempool into the proposal or even ignore all the transactions altogether. What the application does with the transactions provided to it by RequestPrepareProposal have no effect on CometBFT's mempool.
```

Note, that the application defines the semantics of the `PrepareProposal` and it MAY be non-deterministic and is only executed by the current block proposer.

Now, reading mempool twice in the previous sentence is confusing, lets break it down. CometBFT has a mempool that handles gossiping transactions to other nodes in the network. How these transactions are ordered is determined by CometBFT's mempool, typically FIFO. However, since the application is able to fully inspect all transactions, it can provide greater control over transaction ordering. Allowing the application to handle ordering enables the application to define how it would like the block constructed.

The Cosmos SDK defines the `DefaultProposalHandler` type, which provides applications with `PrepareProposal` and `ProcessProposal` handlers. If you decide to implement your own `PrepareProposal` handler, you must be sure to ensure that the transactions selected DO NOT exceed the maximum block gas (if set) and the maximum bytes provided by `req.MaxBytes`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci\_utils.go
```

This default implementation can be overridden by the application developer in favor of a custom implementation in [app.go](#):

```
prepareOpt := func(app *baseapp.BaseApp) {
    abciPropHandler := baseapp.NewDefaultProposalHandler(mempool, app)
    app.SetPrepareProposal(abciPropHandler.PrepareProposalHandler())
}

baseAppOptions = append(baseAppOptions, prepareOpt)
```

Process Proposal

`ProcessProposal` handles the validation of a proposal from `PrepareProposal`, which also includes a block header. Meaning, that after a block has been proposed the other validators have the right to vote on a block. The validator in the default implementation of `PrepareProposal` runs basic validity checks on each transaction.

Note, `ProcessProposal` MAY NOT be non-deterministic, i.e. it must be deterministic. This means if `ProcessProposal` panics or fails and we reject, all honest validator processes will prevote nil and the CometBFT round will proceed again until a valid proposal is proposed.

Here is the implementation of the default implementation:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci\_utils.go#L153-L159
```

Like `PrepareProposal` this implementation is the default and can be modified by the application developer in `app.go`. If you decide to implement your own `ProcessProposal` handler, you must be sure to ensure that the transactions provided in the proposal DO NOT exceed the maximum block gas (if set).

```
processOpt := func(app *baseapp.BaseApp) {
    abciPropHandler := baseapp.NewDefaultProposalHandler(mempool, app)
    app.SetProcessProposal(abciPropHandler.ProcessProposalHandler())
}

baseAppOptions = append(baseAppOptions, processOpt)
```

Mempool

Now that we have walked through the `PrepareProposal` & `ProcessProposal`, we can move on to walking through the mempool.

There are countless designs that an application developer can write for a mempool, the SDK opted to provide only simple mempool implementations. Namely, the SDK provides the following mempools:

- [No-op Mempool](#)
- [Sender Nonce Mempool](#)
- [Priority Nonce Mempool](#)

The default SDK is a [No-op Mempool](#), but it can be replaced by the application developer in `app.go`:

```
nonceMempool := mempool.NewSenderNonceMempool()
mempoolOpt := baseapp.SetMempool(nonceMempool)
baseAppOptions = append(baseAppOptions, mempoolOpt)
```

No-op Mempool

A no-op mempool is a mempool where transactions are completely discarded and ignored when BaseApp interacts with the mempool. When this mempool is used, it assumed that an application will rely on CometBFT's transaction ordering defined in `RequestPrepareProposal`, which is FIFO-ordered by default.

Note: If a NoOp mempool is used, `PrepareProposal` and `ProcessProposal` both should be aware of this as `PrepareProposal` could include transactions that could fail verification in `ProcessProposal`.

Sender Nonce Mempool

The nonce mempool is a mempool that keeps transactions sorted by nonce in order to avoid the issues with nonces. It works by storing the transaction in a list sorted by the transaction nonce. When the proposer asks for transactions to be included in a block it randomly selects a sender and gets the first transaction in the list. It repeats this until the mempool is empty or the block is full.

It is configurable with the following parameters:

MaxTxs

It is an integer value that sets the mempool in one of three modes, *bounded*, *unbounded*, or *disabled*.

- **negative:** Disabled, mempool does not insert new transaction and return early.

- **zero**: Unbounded mempool has no transaction limit and will never fail with `ErrMempoolTxMaxCapacity`.
- **positive**: Bounded, it fails with `ErrMempoolTxMaxCapacity` when `maxTx` value is the same as `CountTx()`

Seed

Set the seed for the random number generator used to select transactions from the mempool.

Priority Nonce Mempool

The [priority nonce mempool](#) is a mempool implementation that stores txs in a partially ordered set by 2 dimensions:

- priority
- sender-nonce (sequence number)

Internally it uses one priority ordered [skip list](#) and one skip list per sender ordered by sender-nonce (sequence number). When there are multiple txs from the same sender, they are not always comparable by priority to other sender txs and must be partially ordered by both sender-nonce and priority.

It is configurable with the following parameters:

MaxTxs

It is an integer value that sets the mempool in one of three modes, *bounded*, *unbounded*, or *disabled*.

- **negative**: Disabled, mempool does not insert new transaction and return early.
- **zero**: Unbounded mempool has no transaction limit and will never fail with `ErrMempoolTxMaxCapacity`.
- **positive**: Bounded, it fails with `ErrMempoolTxMaxCapacity` when `maxTx` value is the same as `CountTx()`

Callback

The priority nonce mempool provides mempool options allowing the application sets callback(s).

- **OnRead**: Set a callback to be called when a transaction is read from the mempool.
- **TxReplacement**: Sets a callback to be called when duplicated transaction nonce detected during mempool insert. Application can define a transaction replacement rule based on tx priority or certain transaction fields.

More information on the SDK mempool implementation can be found in the [godocs](#).

sidebar_position: 1

Application Upgrade

:::note This document describes how to upgrade your application. If you are looking specifically for the changes to perform between SDK versions, see the [SDK migrations documentation](#). :::

:::warning This section is currently incomplete. Track the progress of this document [here](#). :::

:::note

Pre-requisite Reading

- [x/upgrade Documentation](#)

:::

General Workflow

Let's assume we are running v0.38.0 of our software in our testnet and want to upgrade to v0.40.0. How would this look in practice? First of all, we want to finalize the v0.40.0 release candidate and there install a specially named upgrade handler (eg. "testnet-v2" or even "v0.40.0"). An upgrade handler should be defined in a new version of the software to define what migrations to run to migrate from the older version of the software. Naturally, this is app-specific rather than module specific, and must be defined in `app.go`, even if it imports logic from various modules to perform the actions. You can register them with `upgradeKeeper.SetUpgradeHandler` during the app initialization (before starting the abci server), and they serve not only to perform a migration, but also to identify if this is the old or new version (eg. presence of a handler registered for the named upgrade).

Once the release candidate along with an appropriate upgrade handler is frozen, we can have a governance vote to approve this upgrade at some future block height (e.g. 200000). This is known as an `upgrade.Plan`. The v0.38.0 code will not know of this handler, but will continue to run until block 200000, when the plan kicks in at `BeginBlock`. It will check for existence of the handler, and finding it missing, know that it is running the obsolete software, and gracefully exit.

Generally the application binary will restart on exit, but then will execute this `BeginBlocker` again and exit, causing a restart loop. Either the operator can manually install the new software, or you can make use of an external watcher daemon to possibly download and then switch binaries, also potentially doing a backup. The SDK tool for doing such, is called [Cosmovisor](#).

When the binary restarts with the upgraded version (here v0.40.0), it will detect we have registered the "testnet-v2" upgrade handler in the code, and realize it is the new version. It then will run the upgrade handler and *migrate the database in-place*. Once finished, it marks the upgrade as done, and continues processing the rest of the block as normal. Once 2/3 of the voting power has upgraded, the blockchain will immediately resume the consensus mechanism. If the majority of operators add a custom `do-upgrade` script, this should be a matter of minutes and not even require them to be awake at that time.

Integrating With An App

Setup an upgrade Keeper for the app and then define a `BeginBlocker` that calls the upgrade keeper's `BeginBlocker` method:

```
func (app *myApp) BeginBlocker(ctx sdk.Context, req abci.RequestBeginBlock)
(abci.ResponseBeginBlock, error) {
    app.upgradeKeeper.BeginBlocker(ctx, req)
    return abci.ResponseBeginBlock{}, nil
}
```

The app must then integrate the upgrade keeper with its governance module as appropriate. The governance module should call `ScheduleUpgrade` to schedule an upgrade and `ClearUpgradePlan` to cancel a pending upgrade.

Performing Upgrades

Upgrades can be scheduled at a predefined block height. Once this block height is reached, the existing software will cease to process ABCI messages and a new version with code that handles the upgrade must be deployed. All upgrades are coordinated by a unique upgrade name that cannot be reused on the same blockchain. In order for the upgrade module to know that the upgrade has been safely applied, a handler with the name of the upgrade must be installed. Here is an example handler for an upgrade named "my-fancy-upgrade":

```
app.upgradeKeeper.SetUpgradeHandler("my-fancy-upgrade", func(ctx sdk.Context, plan
upgrade.Plan) {
    // Perform any migrations of the state store needed for this upgrade
})
```

This upgrade handler performs the dual function of alerting the upgrade module that the named upgrade has been applied, as well as providing the opportunity for the upgraded software to perform any necessary state migrations. Both the halt (with the old binary) and applying the migration (with the new binary) are enforced in the state machine. Actually switching the binaries is an ops task and not handled inside the sdk / abci app.

Here is a sample code to set store migrations with an upgrade:

```
// this configures a no-op upgrade handler for the "my-fancy-upgrade" upgrade
app.UpgradeKeeper.SetUpgradeHandler("my-fancy-upgrade", func(ctx sdk.Context, plan
upgrade.Plan) {
    // upgrade changes here
})
upgradeInfo, err := app.UpgradeKeeper.ReadUpgradeInfoFromDisk()
if err != nil {
    // handle error
}
if upgradeInfo.Name == "my-fancy-upgrade" &&
!app.UpgradeKeeper.IsSkipHeight(upgradeInfo.Height) {
    storeUpgrades := store.StoreUpgrades{
        Renamed: []store.StoreRename{
            OldKey: "foo",
            NewKey: "bar",
        },
        Deleted: []string{},
    }
    // configure store loader that checks if version == upgradeHeight and applies store
    upgrades
    app.SetStoreLoader(upgrade.UpgradeStoreLoader(upgradeInfo.Height, &storeUpgrades))
}
```

Halt Behavior

Before halting the ABCI state machine in the `BeginBlocker` method, the upgrade module will log an error that looks like:

```
UPGRADE "<Name>" NEEDED at height <NNNN>: <Info>
```

where `Name` and `Info` are the values of the respective fields on the upgrade Plan.

To perform the actual halt of the blockchain, the upgrade keeper simply panics which prevents the ABCI state machine from proceeding but doesn't actually exit the process. Exiting the process can cause issues for other nodes that start to lose connectivity with the exiting nodes, thus this module prefers to just halt but not exit.

Automation

Read more about [Cosmovisor](#), the tool for automating upgrades.

Cancelling Upgrades

There are two ways to cancel a planned upgrade - with on-chain governance or off-chain social consensus. For the first one, there is a `CancelSoftwareUpgrade` governance proposal, which can be voted on and will remove the scheduled upgrade plan. Of course this requires that the upgrade was known to be a bad idea well before the upgrade itself, to allow time for a vote. If you want to allow such a possibility, you should set the upgrade height to be `2 * (votingperiod + depositperiod) + (safety delta)` from the beginning of the first upgrade proposal. Safety delta is the time available from the success of an upgrade proposal and the realization it was a bad idea (due to external testing). You can also start a `CancelSoftwareUpgrade` proposal while the original `SoftwareUpgrade` proposal is still being voted upon, as long as the voting period ends after the `SoftwareUpgrade` proposal.

However, let's assume that we don't realize the upgrade has a bug until shortly before it will occur (or while we try it out - hitting some panic in the migration). It would seem the blockchain is stuck, but we need to allow an escape for social consensus to overrule the planned upgrade. To do so, there's a `--unsafe-skip-upgrades` flag to the start command, which will cause the node to mark the upgrade as done upon hitting the planned upgrade height(s), without halting and without actually performing a migration. If over two-thirds run their nodes with this flag on the old binary, it will allow the chain to continue through the upgrade with a manual override. (This must be well-documented for anyone syncing from genesis later on).

Example:

```
<appd> start --unsafe-skip-upgrades <height1> <optional_height_2> ...  
<optional_height_N>
```

Pre-Upgrade Handling

Cosmovisor supports custom pre-upgrade handling. Use pre-upgrade handling when you need to implement application config changes that are required in the newer version before you perform the upgrade.

Using Cosmovisor pre-upgrade handling is optional. If pre-upgrade handling is not implemented, the upgrade continues.

For example, make the required new-version changes to `app.toml` settings during the pre-upgrade handling. The pre-upgrade handling process means that the file does not have to be manually updated after the upgrade.

Before the application binary is upgraded, Cosmovisor calls a `pre-upgrade` command that can be implemented by the application.

The `pre-upgrade` command does not take in any command-line arguments and is expected to terminate with the following exit codes:

Exit status code	How it is handled in Cosmosvisor
0	Assumes <code>pre-upgrade</code> command executed successfully and continues the upgrade.
1	Default exit code when <code>pre-upgrade</code> command has not been implemented.
30	<code>pre-upgrade</code> command was executed but failed. This fails the entire upgrade.
31	<code>pre-upgrade</code> command was executed but failed. But the command is retried until exit code 1 or 30 are returned.

Sample

Here is a sample structure of the `pre-upgrade` command:

```
func preUpgradeCommand() *cobra.Command {
    cmd := &cobra.Command{
        Use:   "pre-upgrade",
        Short: "Pre-upgrade command",
        Long:  "Pre-upgrade command to implement custom pre-upgrade handling",
        Run: func(cmd *cobra.Command, args []string) {
            err := HandlePreUpgrade()

            if err != nil {
                os.Exit(30)
            }

            os.Exit(0)
        },
    }

    return cmd
}
```

Ensure that the pre-upgrade command has been registered in the application:

```
rootCmd.AddCommand(
    // ..
    preUpgradeCommand(),
    // ..
)
```

When not using Cosmovisor, ensure to run `<appd> pre-upgrade` before starting the application binary.

sidebar_position: 1

Vote Extensions

:::note Synopsis This sections describes how the application can define and use vote extensions defined in ABCI++. :::

Extend Vote

ABCI++ allows an application to extend a pre-commit vote with arbitrary data. This process does NOT have to be deterministic and the data returned can be unique to the validator process. The Cosmos SDK defines `baseapp.ExtendVoteHandler` :

```
type ExtendVoteHandler func(Context, *abci.RequestExtendVote)
(*abci.ResponseExtendVote, error)
```

An application can set this handler in `app.go` via the `baseapp.SetExtendVoteHandler` `BaseApp` option function. The `sdk.ExtendVoteHandler`, if defined, is called during the `ExtendVote` ABCI method. Note, if an application decides to implement `baseapp.ExtendVoteHandler`, it MUST return a non-nil `VoteExtension`. However, the vote extension can be empty. See [here](#) for more details.

There are many decentralized censorship-resistant use cases for vote extensions. For example, a validator may want to submit prices for a price oracle or encryption shares for an encrypted transaction mempool. Note, an application should be careful to consider the size of the vote extensions as they could increase latency in block production. See [here](#) for more details.

Verify Vote Extension

Similar to extending a vote, an application can also verify vote extensions from other validators when validating their pre-commits. For a given vote extension, this process MUST be deterministic. The Cosmos SDK defines `sdk.VerifyVoteExtensionHandler` :

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/abci.go#L26-L27
```

An application can set this handler in `app.go` via the `baseapp.SetVerifyVoteExtensionHandler` `BaseApp` option function. The `sdk.VerifyVoteExtensionHandler`, if defined, is called during the `VerifyVoteExtension` ABCI method. If an application defines a vote extension handler, it should also define a verification handler. Note, not all validators will share the same view of what vote extensions they verify depending on how votes are propagated. See [here](#) for more details.

Vote Extension Propagation

The agreed upon vote extensions at height `H` are provided to the proposing validator at height `H+1` during `PrepareProposal`. As a result, the vote extensions are not natively provided or exposed to the

remaining validators during `ProcessProposal`. As a result, if an application requires that the agreed upon vote extensions from height `H` are available to all validators at `H+1`, the application must propagate these vote extensions manually in the block proposal itself. This can be done by "injecting" them into the block proposal, since the `Txs` field in `PrepareProposal` is just a slice of byte slices.

FinalizeBlock will ignore any byte slice that doesn't implement an `sdk.Tx` so any injected vote extensions will safely be ignored in `FinalizeBlock`. For more details on propagation, see the [ABCI++ 2.0 ADR](#).

sidebar_position: 1

Introduction to Cosmos SDK Modules

:::note Synopsis Modules define most of the logic of Cosmos SDK applications. Developers compose modules together using the Cosmos SDK to build their custom application-specific blockchains. This document outlines the basic concepts behind SDK modules and how to approach module management. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK application](#)
- [Lifecycle of a Cosmos SDK transaction](#)

:::

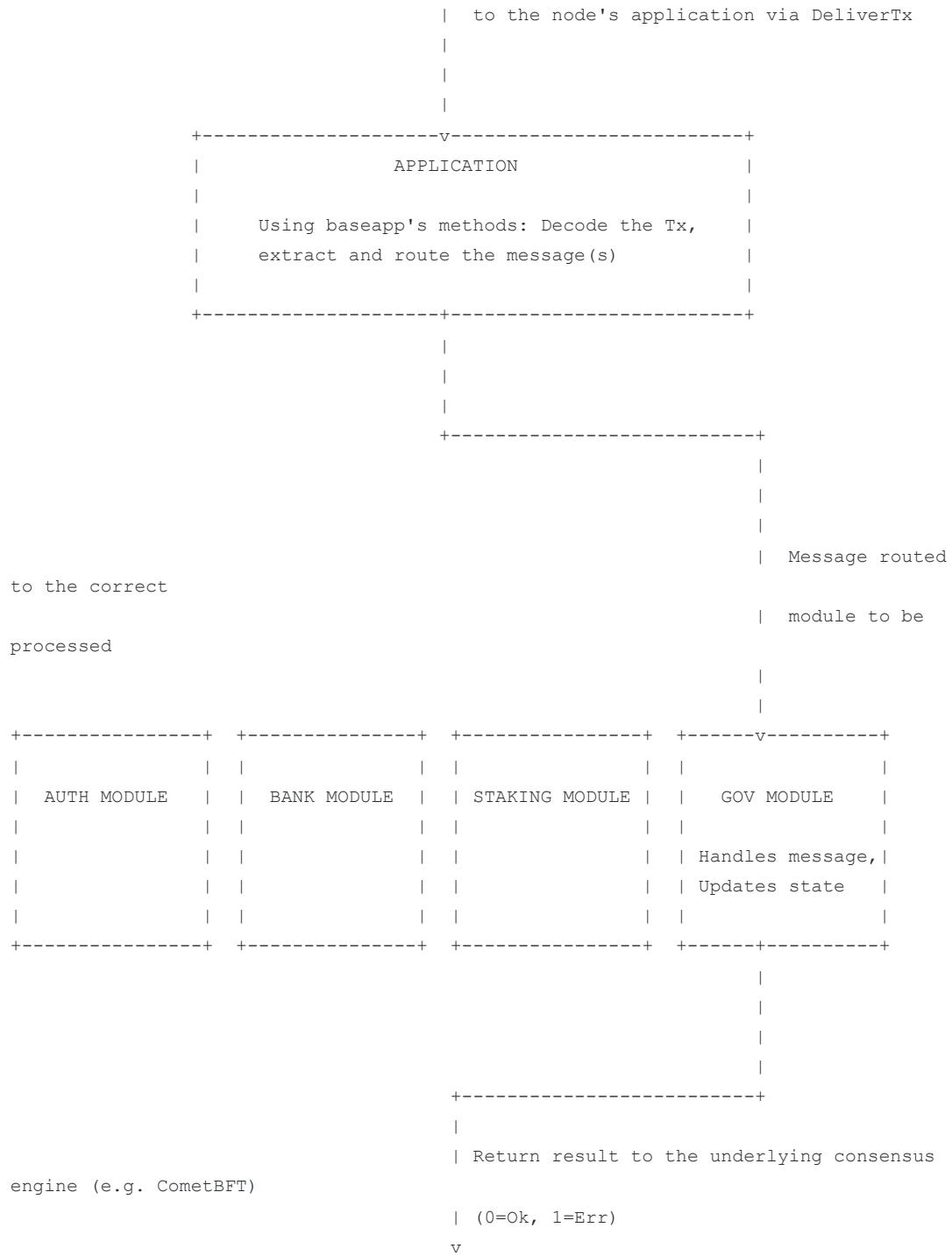
Role of Modules in a Cosmos SDK Application

The Cosmos SDK can be thought of as the Ruby-on-Rails of blockchain development. It comes with a core that provides the basic functionalities every blockchain application needs, like a [boilerplate implementation of the ABCI](#) to communicate with the underlying consensus engine, a [multistore](#) to persist state, a [server](#) to form a full-node and [interfaces](#) to handle queries.

On top of this core, the Cosmos SDK enables developers to build modules that implement the business logic of their application. In other words, SDK modules implement the bulk of the logic of applications, while the core does the wiring and enables modules to be composed together. The end goal is to build a robust ecosystem of open-source Cosmos SDK modules, making it increasingly easier to build complex blockchain applications.

Cosmos SDK modules can be seen as little state-machines within the state-machine. They generally define a subset of the state using one or more `KVStore`s in the [main multistore](#), as well as a subset of [message types](#). These messages are routed by one of the main components of Cosmos SDK core, `BaseApp`, to a module Protobuf `Msg service` that defines them.

```
+  
|  
|   Transaction relayed from the full-node's  
consensus engine
```



As a result of this architecture, building a Cosmos SDK application usually revolves around writing modules to implement the specialized logic of the application and composing them with existing modules to complete the application. Developers will generally work on modules that implement logic needed for their specific use case that do not exist yet, and will use existing modules for more generic functionalities like staking, accounts, or token management.

How to Approach Building Modules as a Developer

While there are no definitive guidelines for writing modules, here are some important design principles developers should keep in mind when building them:

- **Composability:** Cosmos SDK applications are almost always composed of multiple modules. This means developers need to carefully consider the integration of their module not only with the core of the Cosmos SDK, but also with other modules. The former is achieved by following standard design patterns outlined [here](#), while the latter is achieved by properly exposing the store(s) of the module via the `keeper`.
- **Specialization:** A direct consequence of the **composability** feature is that modules should be **specialized**. Developers should carefully establish the scope of their module and not batch multiple functionalities into the same module. This separation of concerns enables modules to be re-used in other projects and improves the upgradability of the application. **Specialization** also plays an important role in the [object-capabilities model](#) of the Cosmos SDK.
- **Capabilities:** Most modules need to read and/or write to the store(s) of other modules. However, in an open-source environment, it is possible for some modules to be malicious. That is why module developers need to carefully think not only about how their module interacts with other modules, but also about how to give access to the module's store(s). The Cosmos SDK takes a capabilities-oriented approach to inter-module security. This means that each store defined by a module is accessed by a `key`, which is held by the module's `keeper`. This `keeper` defines how to access the store(s) and under what conditions. Access to the module's store(s) is done by passing a reference to the module's `keeper`.

Main Components of Cosmos SDK Modules

Modules are by convention defined in the `./x/` subfolder (e.g. the `bank` module will be defined in the `./x/bank` folder). They generally share the same core components:

- A `keeper`, used to access the module's store(s) and update the state.
- A `Msg service`, used to process messages when they are routed to the module by `BaseApp` and trigger state-transitions.
- A `query service`, used to process user queries when they are routed to the module by `BaseApp`.
- Interfaces, for end users to query the subset of the state defined by the module and create `message`s of the custom types defined in the module.

In addition to these components, modules implement the `AppModule` interface in order to be managed by the [module manager](#).

Please refer to the [structure document](#) to learn about the recommended structure of a module's directory.

sidebar_position: 1

Module Manager

:::note Synopsis Cosmos SDK modules need to implement the `AppModule interfaces`, in order to be managed by the application's [module manager](#). The module manager plays an important role in [message and query routing](#), and allows application developers to set the order of execution of a variety of functions like [BeginBlocker](#) and [EndBlocker](#). :::

:::note

Pre-requisite Readings

- [Introduction to Cosmos SDK Modules](#)

:::

Application Module Interfaces

Application module interfaces exist to facilitate the composition of modules together to form a functional Cosmos SDK application. There are 4 main application module interfaces:

- `AppModuleBasic` for independent module functionalities.
- `AppModule` for inter-dependent module functionalities (except genesis-related functionalities).
- `AppModuleGenesis` for inter-dependent genesis-related module functionalities.
- `GenesisOnlyAppModule` : Defines an `AppModule` that only has import/export functionality

The above interfaces are mostly embedding smaller interfaces (extension interfaces), that defines specific functionalities:

- `HasName` : Allows the module to provide its own name for legacy purposes.
- `HasGenesisBasics` : The legacy interface for stateless genesis methods.
- `HasGenesis` : The extension interface for stateful genesis methods.
- `HasInvariants` : The extension interface for registering invariants.
- `HasServices` : The extension interface for modules to register services.
- `HasConsensusVersion` : The extension interface for declaring a module consensus version.
- `HasBeginBlocker` : The extension interface that contains information about the `AppModule` and `BeginBlock`.
- `HasEndBlocker` : The extension interface that contains information about the `AppModule` and `EndBlock`.
- `HasABCIEndblock` : The extension interface that contains information about the `AppModule`, `EndBlock` and returns an updated validator set.
- `HasPrecommit` : The extension interface that contains information about the `AppModule` and `Precommit`.
- `HasPrepareCheckState` : The extension interface that contains information about the `AppModule` and `PrepareCheckState`.

The `AppModuleBasic` interface exists to define independent methods of the module, i.e. those that do not depend on other modules in the application. This allows for the construction of the basic application structure early in the application definition, generally in the `init()` function of the [main application file](#).

The `AppModule` interface exists to define inter-dependent module methods. Many modules need to interact with other modules, typically through `keeper s`, which means there is a need for an interface where modules list their `keeper s` and other methods that require a reference to another module's object. `AppModule` interface extension, such as `HasBeginBlocker` and `HasEndBlocker`, also enables the module manager to set the order of execution between module's methods like `BeginBlock` and `EndBlock`, which is important in cases where the order of execution between modules matters in the context of the application.

The usage of extension interfaces allows modules to define only the functionalities they need. For example, a module that does not need an `EndBlock` does not need to define the `HasEndBlocker` interface and

thus the `EndBlock` method. `AppModule` and `AppModuleGenesis` are voluntarily small interfaces, that can take advantage of the `Module` patterns without having to define many placeholder functions.

AppModuleBasic

The `AppModuleBasic` interface defines the independent methods modules need to implement.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/types/module/module.go#L56-L66
```

Let us go through the methods:

- `RegisterLegacyAminoCodec(*codec.LegacyAmino)` : Registers the `amino` codec for the module, which is used to marshal and unmarshal structs to/from `[]byte` in order to persist them in the module's `KVStore`.
- `RegisterInterfaces(codectypes.InterfaceRegistry)` : Registers a module's interface types and their concrete implementations as `proto.Message`.
- `RegisterGRPCGatewayRoutes(client.Context, *runtime.ServeMux)` : Registers gRPC routes for the module.

All the `AppModuleBasic` of an application are managed by the [BasicManager](#).

HasName

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/types/module/module.go#L71-L73
```

- `HasName` is an interface that has a method `Name()`. This method returns the name of the module as a `string`.

HasGenesisBasics

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/types/module/module.go#L76-L79
```

Let us go through the methods:

- `DefaultGenesis(codec.JSONCodec)` : Returns a default [GenesisState](#) for the module, marshalled to `json.RawMessage`. The default `GenesisState` need to be defined by the module developer and is primarily used for testing.
- `ValidateGenesis(codec.JSONCodec, client.TxEncodingConfig, json.RawMessage)` : Used to validate the `GenesisState` defined by a module, given in its `json.RawMessage` form. It will usually unmarshal the `json` before running a custom [ValidateGenesis](#) function defined by the module developer.

AppModuleGenesis

The `AppModuleGenesis` interface is a simple embedding of the `AppModuleBasic` and `HasGenesis` interfaces.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L183-L186
```

It does not have its own manager, and exists separately from [AppModule](#) only for modules that exist only to implement genesis functionalities, so that they can be managed without having to implement all of [AppModule](#)'s methods.

HasGenesis

The `HasGenesis` interface is an extension interface of `HasGenesisBasics`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/core/appmodule/genesis.go#L10-L24
```

Let us go through the two added methods:

- `InitGenesis(sdk.Context, codec.JSONCodec, json.RawMessage)` : Initializes the subset of the state managed by the module. It is called at genesis (i.e. when the chain is first started).
- `ExportGenesis(sdk.Context, codec.JSONCodec)` : Exports the latest subset of the state managed by the module to be used in a new genesis file. `ExportGenesis` is called for each module when a new chain is started from the state of an existing chain.

AppModule

The `AppModule` interface defines a module. Modules can declare their functionalities by implementing extensions interfaces.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L197-L199
```

`AppModule`s are managed by the [module manager](#), which checks which extension interfaces are implemented by the module.

:::note Previously the `AppModule` interface was containing all the methods that are defined in the extensions interfaces. This was leading to much boilerplate for modules that did not need all the functionalities. :::

HasInvariants

This interface defines one method. It allows to checks if a module can register invariants.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L202-L205
```

- `RegisterInvariants(sdk.InvariantRegistry)` : Registers the [invariants](#) of the module. If an invariant deviates from its predicted value, the [InvariantRegistry](#) triggers appropriate logic (most often the chain will be halted).

HasServices

This interface defines one method. It allows to checks if a module can register invariants.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L208-L211
```

- `RegisterServices(Configurator)` : Allows a module to register services.

HasConsensusVersion

This interface defines one method for checking a module consensus version.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L214-L220
```

- `ConsensusVersion() uint64` : Returns the consensus version of the module.

HasBeginBlocker

The `HasBeginBlocker` is an extension interface from `AppModule`. All modules that have an `BeginBlock` method implement this interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/core/appmodule/module.go#L56-L63
```

- `BeginBlock(sdk.Context) error` : This method gives module developers the option to implement logic that is automatically triggered at the beginning of each block.

HasEndBlocker

The `HasEndBlocker` is an extension interface from `AppModule`. All modules that have an `EndBlock` method implement this interface. If a module need to return validator set updates (staking), they can use `HasABCIEndblock`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/core/appmodule/module.go#L66-L72
```

- `EndBlock(sdk.Context) error` : This method gives module developers the option to implement logic that is automatically triggered at the end of each block.

HasABCIEndblock

The `HasABCIEndblock` is an extension interface from `AppModule`. All modules that have an `EndBlock` which return validator set updates implement this interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L222-L225
```

- `EndBlock(sdk.Context) ([]abci.ValidatorUpdate, error)` : This method gives module developers the option to inform the underlying consensus engine of validator set changes (e.g. the `staking` module).

HasPrecommit

`HasPrecommit` is an extension interface from `AppModule`. All modules that have a `Precommit` method implement this interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/core/appmodule/module.go#L49-L52
```

- `Precommit(sdk.Context)` : This method gives module developers the option to implement logic that is automatically triggered during `[Commit'](.../core/00-baseapp.md#commit)` of each block using the `[finalizeblockstate](.../core/00-baseapp.md#state-updates)` of the block to be committed. Implement empty if no logic needs to be triggered during `Commit`` of each block for this module.

`HasPrepareCheckState`

`HasPrepareCheckState` is an extension interface from `AppModule`. All modules that have a `PrepareCheckState` method implement this interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/core/appmodule/module.go#L49-L52
```

- `PrepareCheckState(sdk.Context)` : This method gives module developers the option to implement logic that is automatically triggered during `[Commit'](.../core/00-baseapp.md#commit)` of each block using the `[checkState](.../core/00-baseapp.md#state-updates)` of the next block. Implement empty if no logic needs to be triggered during `Commit`` of each block for this module.

Implementing the Application Module Interfaces

Typically, the various application module interfaces are implemented in a file called `module.go`, located in the module's folder (e.g. `./x/module/module.go`).

Almost every module needs to implement the `AppModuleBasic` and `AppModule` interfaces. If the module is only used for genesis, it will implement `AppModuleGenesis` instead of `AppModule`. The concrete type that implements the interface can add parameters that are required for the implementation of the various methods of the interface. For example, the `Route()` function often calls a `NewMsgServerImpl(k keeper)` function defined in `keeper/msg_server.go` and therefore needs to pass the module's [keeper](#) as a parameter.

```
// example
type AppModule struct {
    AppModuleBasic
    keeper      Keeper
}
```

In the example above, you can see that the `AppModule` concrete type references an `AppModuleBasic`, and not an `AppModuleGenesis`. That is because `AppModuleGenesis` only needs to be implemented in modules that focus on genesis-related functionalities. In most modules, the concrete `AppModule` type will have a reference to an `AppModuleBasic` and implement the two added methods of `AppModuleGenesis` directly in the `AppModule` type.

If no parameter is required (which is often the case for `AppModuleBasic`), just declare an empty concrete type like so:

```
type AppModuleBasic struct{}
```

Module Managers

Module managers are used to manage collections of `AppModuleBasic` and `AppModule`.

BasicManager

The `BasicManager` is a structure that lists all the `AppModuleBasic` of an application:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L82
```

It implements the following methods:

- `NewBasicManager(modules ...AppModuleBasic)` : Constructor function. It takes a list of the application's `AppModuleBasic` and builds a new `BasicManager`. This function is generally called in the `init()` function of [app.go](#) to quickly initialize the independent elements of the application's modules (click [here](#) to see an example).
- `RegisterLegacyAminoCodec(cdc *codec.LegacyAmino)` : Registers the `codec.LegacyAmino`s of each of the application's `AppModuleBasic`. This function is usually called early on in the [application's construction](#).
- `RegisterInterfaces(registry codectypes.InterfaceRegistry)` : Registers interface types and implementations of each of the application's `AppModuleBasic`.
- `DefaultGenesis(cdc codec.JSONCodec)` : Provides default genesis information for modules in the application by calling the [DefaultGenesis\(codec.JSONCodec\)](#) function of each module. It only calls the modules that implements the `HasGenesisBasics` interfaces.
- `ValidateGenesis(cdc codec.JSONCodec, txEncCfg client.TxEncodingConfig, genesis map[string]json.RawMessage)` : Validates the genesis information modules by calling the [ValidateGenesis\(codec.JSONCodec, client.TxEncodingConfig, json.RawMessage\)](#) function of modules implementing the `HasGenesisBasics` interface.
- `RegisterGRPCGatewayRoutes(clientCtx client.Context, rtr *runtime.ServeMux)` : Registers gRPC routes for modules.
- `AddTxCommands(rootTxCmd *cobra.Command)` : Adds modules' transaction commands (defined as `GetTxCmd()` *`cobra.Command`) to the application's [rootTxCommand](#). This function is usually called function from the `main.go` function of the [application's command-line interface](#).
- `AddQueryCommands(rootQueryCmd *cobra.Command)` : Adds modules' query commands (defined as `GetQueryCmd()` *`cobra.Command`) to the application's [rootQueryCommand](#). This function is usually called function from the `main.go` function of the [application's command-line interface](#).

Manager

The `Manager` is a structure that holds all the `AppModule` of an application, and defines the order of execution between several key components of these modules:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/module/module.go#L267-L276
```

The module manager is used throughout the application whenever an action on a collection of modules is required. It implements the following methods:

- `NewManager(modules ... AppModule)` : Constructor function. It takes a list of the application's `AppModule`s and builds a new `Manager`. It is generally called from the application's main [constructor function](#).
- `SetOrderInitGenesis(moduleNames ...string)` : Sets the order in which the [`InitGenesis`](#) function of each module will be called when the application is first started. This function is generally called from the application's main [constructor function](#). To initialize modules successfully, module dependencies should be considered. For example, the `genutil` module must occur after `staking` module so that the pools are properly initialized with tokens from genesis accounts, the `genutils` module must also occur after `auth` so that it can access the params from auth, IBC's `capability` module should be initialized before all other modules so that it can initialize any capabilities.
- `SetOrderExportGenesis(moduleNames ...string)` : Sets the order in which the [`ExportGenesis`](#) function of each module will be called in case of an export. This function is generally called from the application's main [constructor function](#).
- `SetOrderBeginBlockers(moduleNames ...string)` : Sets the order in which the `BeginBlock()` function of each module will be called at the beginning of each block. This function is generally called from the application's main [constructor function](#).
- `SetOrderEndBlockers(moduleNames ...string)` : Sets the order in which the `EndBlock()` function of each module will be called at the end of each block. This function is generally called from the application's main [constructor function](#).
- `SetOrderPrecommitters(moduleNames ...string)` : Sets the order in which the `Precommit()` function of each module will be called during commit of each block. This function is generally called from the application's main [constructor function](#).
- `SetOrderPrepareCheckStakers(moduleNames ...string)` : Sets the order in which the `PrepareCheckState()` function of each module will be called during commit of each block. This function is generally called from the application's main [constructor function](#).
- `SetOrderMigrations(moduleNames ...string)` : Sets the order of migrations to be run. If not set then migrations will be run with an order defined in `DefaultMigrationsOrder`.
- `RegisterInvariants(ir sdk.InvariantRegistry)` : Registers the [`invariants`](#) of module implementing the `HasInvariants` interface.
- `RegisterRoutes(router sdk.Router, queryRouter sdk.QueryRouter, legacyQuerierCdc *codec.LegacyAmino)` : Registers legacy [`Msg`](#) and [`querier`](#) routes.
- `RegisterServices(cfg Configurator)` : Registers the services of modules implementing the `HasServices` interface.
- `InitGenesis(ctx sdk.Context, cdc codec.JSONCodec, genesisData map[string]json.RawMessage)` : Calls the [`InitGenesis`](#) function of each module when the application is first started, in the order defined in `OrderInitGenesis`. Returns an `abci.ResponseInitChain` to the underlying consensus engine, which can contain validator updates.
- `ExportGenesis(ctx sdk.Context, cdc codec.JSONCodec)` : Calls the [`ExportGenesis`](#) function of each module, in the order defined in `OrderExportGenesis`. The export constructs a

genesis file from a previously existing state, and is mainly used when a hard-fork upgrade of the chain is required.

- `ExportGenesisForModules(ctx sdk.Context, cdc codec.JSONCodec, modulesToExport []string)` : Behaves the same as `ExportGenesis`, except takes a list of modules to export.
- `BeginBlock(ctx sdk.Context) error` : At the beginning of each block, this function is called from `BaseApp` and, in turn, calls the `BeginBlock` function of each modules implementing the `BeginBlock AppModule` interface, in the order defined in `OrderBeginBlockers`. It creates a child `context` with an event manager to aggregate `events` emitted from all modules.
- `EndBlock(ctx sdk.Context) error` : At the end of each block, this function is called from `BaseApp` and, in turn, calls the `EndBlock` function of each modules implementing the `HasEndBlocker` interface, in the order defined in `OrderEndBlockers`. It creates a child `context` with an event manager to aggregate `events` emitted from all modules. The function returns an `abci` which contains the aforementioned events, as well as validator set updates (if any).
- `EndBlock(context.Context) ([]abci.ValidatorUpdate, error)` : At the end of each block, this function is called from `BaseApp` and, in turn, calls the `EndBlock` function of each modules implementing the `HasEndBlocker` interface, in the order defined in `OrderEndBlockers`. It creates a child `context` with an event manager to aggregate `events` emitted from all modules. The function returns an `abci` which contains the aforementioned events, as well as validator set updates (if any).
- `Precommit(ctx sdk.Context)` : During `Commit`, this function is called from `BaseApp` immediately before the `deliverState` is written to the underlying `rootMultiStore` and, in turn calls the `Precommit` function of each modules implementing the `HasPrecommit` interface, in the order defined in `OrderPrecommitters`. It creates a child `context` where the underlying `CacheMultiStore` is that of the newly committed block's `finalizeblockstate`.
- `PrepareCheckState(ctx sdk.Context)` : During `Commit`, this function is called from `BaseApp` immediately after the `deliverState` is written to the underlying `rootMultiStore` and, in turn calls the `PrepareCheckState` function of each module implementing the `HasPrepareCheckState` interface, in the order defined in `OrderPrepareCheckStaters`. It creates a child `context` where the underlying `CacheMultiStore` is that of the next block's `checkState`. Writes to this state will be present in the `checkState` of the next block, and therefore this method can be used to prepare the `checkState` for the next block.

Here's an example of a concrete integration within an `simapp` :

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L411-L434
```

This is the same example from `runtime` (the package that powers app v2):

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/runtime/module.go#L61
```

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/runtime/module.go#L82
```

sidebar_position: 1

Messages and Queries

:::note Synopsis `Msg` s and `Queries` are the two primary objects handled by modules. Most of the core components defined in a module, like `Msg` services, `keeper` s and `Query` services, exist to process message s and queries .:::

:::note

Pre-requisite Readings

- [Introduction to Cosmos SDK Modules](#)

:::

Messages

`Msg` s are objects whose end-goal is to trigger state-transitions. They are wrapped in [transactions](#), which may contain one or more of them.

When a transaction is relayed from the underlying consensus engine to the Cosmos SDK application, it is first decoded by [BaseApp](#) . Then, each message contained in the transaction is extracted and routed to the appropriate module via `BaseApp` 's `MsgServiceRouter` so that it can be processed by the module's [Msg service](#). For a more detailed explanation of the lifecycle of a transaction, click [here](#).

`Msg` Services

Defining Protobuf `Msg` services is the recommended way to handle messages. A Protobuf `Msg` service should be created for each module, typically in `tx.proto` (see more info about [conventions and naming](#)). It must have an RPC service method defined for each message in the module.

See an example of a `Msg` service definition from `x/bank` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/proto/cosmos/bank/v1beta1/tx.proto#L13-L36
```

Each `Msg` service method must have exactly one argument, which must implement the `sdk.Msg` interface, and a Protobuf response. The naming convention is to call the RPC argument `Msg<service-rpc-name>` and the RPC response `Msg<service-rpc-name>Response` . For example:

```
rpc Send(MsgSend) returns (MsgSendResponse);
```

`sdk.Msg` interface is a simplified version of the Amino `LegacyMsg` interface described [below](#) with the `GetSigners()` method. For backwards compatibility with [Amino LegacyMsg](#) s, existing `LegacyMsg` types should be used as the request parameter for service RPC definitions. Newer `sdk.Msg` types, which only support `service` definitions, should use canonical `Msg...` name.

The Cosmos SDK uses Protobuf definitions to generate client and server code:

- `MsgServer` interface defines the server API for the `Msg` service and its implementation is described as part of the [Msg services](#) documentation.

- Structures are generated for all RPC request and response types.

A `RegisterMsgServer` method is also generated and should be used to register the module's `MsgServer` implementation in `RegisterServices` method from the [AppModule interface](#).

In order for clients (CLI and grpc-gateway) to have these URLs registered, the Cosmos SDK provides the function `RegisterMsgServiceDesc(registry codectypes.InterfaceRegistry, sd *grpc.ServiceDesc)` that should be called inside module's [RegisterInterfaces](#) method, using the proto-generated `&_Msg_serviceDesc` as `*grpc.ServiceDesc` argument.

Legacy Amino `LegacyMsg`s

The following way of defining messages is deprecated and using [Msg services](#) is preferred.

Amino `LegacyMsg`s can be defined as protobuf messages. The messages definition usually includes a list of parameters needed to process the message that will be provided by end-users when they want to create a new transaction containing said message.

A `LegacyMsg` is typically accompanied by a standard constructor function, that is called from one of the [module's interface](#). `message`s also need to implement the `sdk.Msg` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/tx_msg.go#L21-L28
```

It extends `proto.Message` and contains the following methods:

- `GetSignBytes() []byte` : Return the canonical byte representation of the message. Used to generate a signature.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/migrations/legacytx/stdsign.go#L21-L29
```

See an example implementation of a `message` from the `gov` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/gov/types/v1 msgs.go#L103-L150
```

Queries

A `query` is a request for information made by end-users of applications through an interface and processed by a full-node. A `query` is received by a full-node through its consensus engine and relayed to the application via the ABCI. It is then routed to the appropriate module via `BaseApp`'s `QueryRouter` so that it can be processed by the module's query service (`./04-query-services.md`). For a deeper look at the lifecycle of a `query`, click [here](#).

gRPC Queries

Queries should be defined using [Protobuf services](#). A `Query` service should be created per module in `query.proto`. This service lists endpoints starting with `rpc`.

Here's an example of such a `Query` service definition:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/auth/v1beta1/query.proto#L14-L89
```

As `proto.Message`s, generated `Response` types implement by default `String()` method of [`fmt.Stringer`](#).

A `RegisterQueryServer` method is also generated and should be used to register the module's query server in the `RegisterServices` method from the [AppModule interface](#).

Legacy Queries

Before the introduction of Protobuf and gRPC in the Cosmos SDK, there was usually no specific `query` object defined by module developers, contrary to `message`s. Instead, the Cosmos SDK took the simpler approach of using a simple `path` to define each `query`. The `path` contains the `query` type and all the arguments needed to process it. For most module queries, the `path` should look like the following:

```
queryCategory/queryRoute/queryType/arg1/arg2/...
```

where:

- `queryCategory` is the category of the `query`, typically `custom` for module queries. It is used to differentiate between different kinds of queries within `BaseApp`'s [Query method](#).
- `queryRoute` is used by `BaseApp`'s [queryRouter](#) to map the `query` to its module. Usually, `queryRoute` should be the name of the module.
- `queryType` is used by the module's [querier](#) to map the `query` to the appropriate `querier` function within the module.
- `args` are the actual arguments needed to process the `query`. They are filled out by the end-user. Note that for bigger queries, you might prefer passing arguments in the `Data` field of the request `req` instead of the `path`.

The `path` for each `query` must be defined by the module developer in the module's [command-line interface file](#). Overall, there are 3 mains components module developers need to implement in order to make the subset of the state defined by their module queryable:

- A [querier](#), to process the `query` once it has been [routed to the module](#).
- [Query commands](#) in the module's CLI file, where the `path` for each `query` is specified.
- `query` return types. Typically defined in a file `types/querier.go`, they specify the result type of each of the module's `queries`. These custom types must implement the `String()` method of [`fmt.Stringer`](#).

Store Queries

Store queries query directly for store keys. They use `clientCtx.QueryABCI(req abci.RequestQuery)` to return the full `abci.ResponseQuery` with inclusion Merkle proofs.

See following examples:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci.go#L864-L894
```

sidebar_position: 1

Msg Services

:::note Synopsis A Protobuf `Msg` service processes [messages](#). Protobuf `Msg` services are specific to the module in which they are defined, and only process messages defined within the said module. They are called from `BaseApp` during [`DeliverTx`](#) . :::

:::note

Pre-requisite Readings

- [Module Manager](#)
- [Messages and Queries](#)

:::

Implementation of a module `Msg` service

Each module should define a Protobuf `Msg` service, which will be responsible for processing requests (implementing `sdk.Msg`) and returning responses.

As further described in [ADR 031](#), this approach has the advantage of clearly specifying return types and generating server and client code.

Protobuf generates a `MsgServer` interface based on a definition of `Msg` service. It is the role of the module developer to implement this interface, by implementing the state transition logic that should happen upon receipt of each `sdk.Msg`. As an example, here is the generated `MsgServer` interface for `x/bank`, which exposes two `sdk.Msg`s:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/bank/types/tx.pb.go#L550-L568
```

When possible, the existing module's `Keeper` should implement `MsgServer`, otherwise a `msgServer` struct that embeds the `Keeper` can be created, typically in `./keeper/msg_server.go`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/bank/keeper/msg_server.go#L17-L19
```

`msgServer` methods can retrieve the `sdk.Context` from the `context.Context` parameter method using the `sdk.UnwrapSDKContext`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/bank/keeper/msg_server.go#L56
```

`sdk.Msg` processing usually follows these 3 steps:

Validation

The message server must perform all validation required (both *stateful* and *stateless*) to make sure the `message` is valid. The `signer` is charged for the gas cost of this validation.

For example, a `msgServer` method for a `transfer` message should check that the sending account has enough funds to actually perform the transfer.

It is recommended to implement all validation checks in a separate function that passes state values as arguments. This implementation simplifies testing. As expected, expensive validation functions charge additional gas. Example:

```
ValidateMsgA(msg MsgA, now Time, gm GasMeter) error {
    if now.Before(msg.Expire) {
        return sdkerrors.ErrInvalidRequest.Wrap("msg expired")
    }
    gm.ConsumeGas(1000, "signature verification")
    return signatureVerificaton(msg.Prover, msg.Data)
}
```

:::warning Previously, the `validateBasic` method was used to perform simple and stateless validation checks. This way of validating is deprecated, this means the `msgServer` must perform all validation checks. :::

State Transition

After the validation is successful, the `msgServer` method uses the `keeper` functions to access the state and perform a state transition.

Events

Before returning, `msgServer` methods generally emit one or more [events](#) by using the `EventManager` held in the `ctx`. Use the new `EmitTypedEvent` function that uses protobuf-based event types:

```
ctx.EventManager().EmitTypedEvent(
    &group.EventABC{Key1: Value1, Key2, Value2})
```

or the older `EmitEvent` function:

```
ctx.EventManager().EmitEvent(
    sdk.NewEvent(
        eventType, // e.g. sdk.EventTypeMessage for a message,
        types.CustomEventType for a custom event defined in the module
        sdk.NewAttribute(key1, value1),
        sdk.NewAttribute(key2, value2),
    ),
)
```

These events are relayed back to the underlying consensus engine and can be used by service providers to implement services around the application. Click [here](#) to learn more about events.

The invoked `msgServer` method returns a `proto.Message` `response` and an `error`. These return values are then wrapped into an `*sdk.Result` or an `error` using `sdk.WrapServiceResult(ctx`

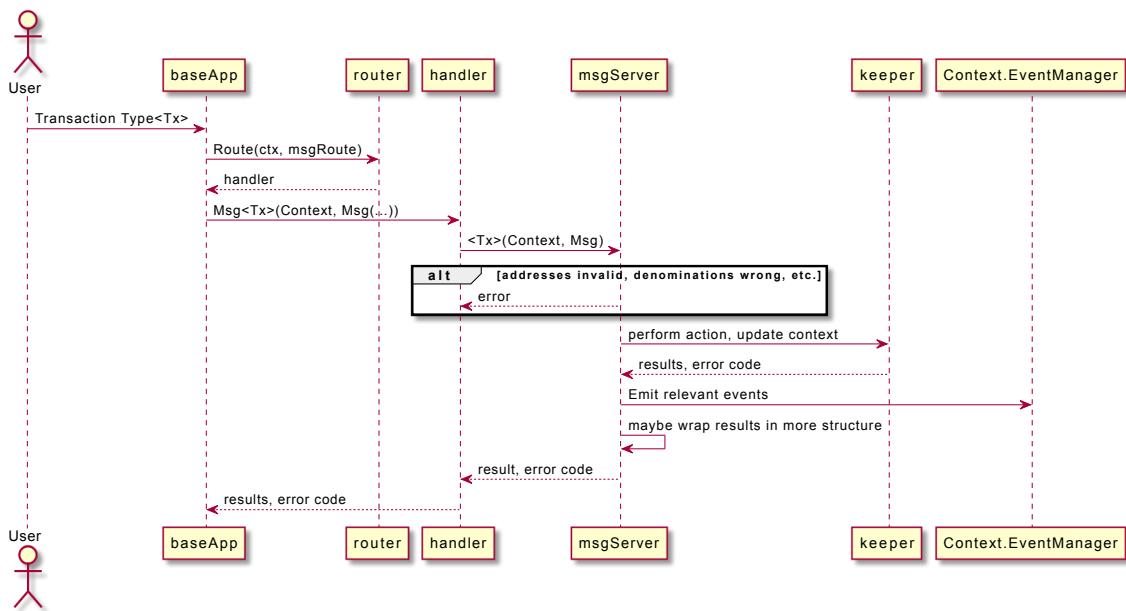
```
sdk.Context, res proto.Message, err error) :
```

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/msg\_service\_router.go#L160
```

This method takes care of marshaling the `res` parameter to protobuf and attaching any events on the `ctx.EventManager()` to the `sdk.Result`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/base/v1beta1/abci.proto#L93-L113
```

This diagram shows a typical structure of a Protobuf `Msg` service, and how the message propagates through the module.



Telemetry

New [telemetry metrics](#) can be created from `msgServer` methods when handling messages.

This is an example from the `x/auth/vesting` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/vesting/msg\_server.go#L76-L88
```

sidebar_position: 1

Query Services

:::note Synopsis A Protobuf Query service processes `queries`. Query services are specific to the module in which they are defined, and only process `queries` defined within said module. They are called from `BaseApp`'s `Query` `method`. :::

:::note

Pre-requisite Readings

- [Module Manager](#)
- [Messages and Queries](#)

:::

Implementation of a module query service

gRPC Service

When defining a Protobuf `Query` service, a `QueryServer` interface is generated for each module with all the service methods:

```
type QueryServer interface {
    QueryBalance(context.Context, *QueryBalanceParams) (*types.Coin, error)
    QueryAllBalances(context.Context, *QueryAllBalancesParams)
    (*QueryAllBalancesResponse, error)
}
```

These custom queries methods should be implemented by a module's keeper, typically in `./keeper/grpc_query.go`. The first parameter of these methods is a generic `context.Context`. Therefore, the Cosmos SDK provides a function `sdk.UnwrapSDKContext` to retrieve the `sdk.Context` from the provided `context.Context`.

Here's an example implementation for the bank module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/keeper/grpc\_query.go
```

Calling queries from the State Machine

The Cosmos SDK v0.47 introduces a new `cosmos.query.v1.module_query_safe` Protobuf annotation which is used to state that a query that is safe to be called from within the state machine, for example:

- a Keeper's query function can be called from another module's Keeper,
- ADR-033 intermodule query calls,
- CosmWasm contracts can also directly interact with these queries.

If the `module_query_safe` annotation set to `true`, it means:

- The query is deterministic: given a block height it will return the same response upon multiple calls, and doesn't introduce any state-machine breaking changes across SDK patch versions.
- Gas consumption never fluctuates across calls and across patch versions.

If you are a module developer and want to use `module_query_safe` annotation for your own query, you have to ensure the following things:

- the query is deterministic and won't introduce state-machine-breaking changes without coordinated upgrades
 - it has its gas tracked, to avoid the attack vector where no gas is accounted for on potentially high-computation queries.
-

sidebar_position: 1

BeginBlocker and EndBlocker

:::note Synopsis `BeginBlocker` and `EndBlocker` are optional methods module developers can implement in their module. They will be triggered at the beginning and at the end of each block respectively, when the `BeginBlock` and `EndBlock` ABCI messages are received from the underlying consensus engine. :::

:::note

Pre-requisite Readings

- [Module Manager](#)

:::

BeginBlocker and EndBlocker

`BeginBlocker` and `EndBlocker` are a way for module developers to add automatic execution of logic to their module. This is a powerful tool that should be used carefully, as complex automatic functions can slow down or even halt the chain.

In 0.47.0, Prepare and Process Proposal were added that allow app developers to do arbitrary work at those phases, but they do not influence the work that will be done in `BeginBlock`. If an application required `BeginBlock` to execute prior to any sort of work is done then this is not possible today (0.50.0).

When needed, `BeginBlocker` and `EndBlocker` are implemented as part of the [`HasBeginBlocker`](#), [`HasABCIEndblocker`](#) and [`EndBlocker` interfaces](#). This means either can be left-out if not required. The `BeginBlock` and `EndBlock` methods of the interface implemented in `module.go` generally defer to `BeginBlocker` and `EndBlocker` methods respectively, which are usually implemented in `abci.go`.

The actual implementation of `BeginBlocker` and `EndBlocker` in `abci.go` are very similar to that of a [`Msg` service](#):

- They generally use the `keeper` and `ctx` to retrieve information about the latest state.
- If needed, they use the `keeper` and `ctx` to trigger state-transitions.
- If needed, they can emit `events` via the `ctx`'s `EventManager`.

A specific type of `EndBlocker` is available to return validator updates to the underlying consensus engine in the form of an [`\[\].abci.ValidatorUpdates`](#). This is the preferred way to implement custom validator changes.

It is possible for developers to define the order of execution between the `BeginBlocker` / `EndBlocker` functions of each of their application's modules via the module's manager `SetOrderBeginBlocker` / `SetOrderEndBlocker` methods. For more on the module manager, click [here](#).

See an example implementation of `BeginBlocker` from the `distribution` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/distribution/abci.go#L14-L38
```

and an example implementation of `EndBlocker` from the `staking` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/staking/keeper/abci.go#L22-L27
```

sidebar_position: 1

Keepers

:::note Synopsis `Keeper`s refer to a Cosmos SDK abstraction whose role is to manage access to the subset of the state defined by various modules. `Keeper`s are module-specific, i.e. the subset of state defined by a module can only be accessed by a `keeper` defined in said module. If a module needs to access the subset of state defined by another module, a reference to the second module's internal `keeper` needs to be passed to the first one. This is done in `app.go` during the instantiation of module keepers. :::

:::note

Pre-requisite Readings

- [Introduction to Cosmos SDK Modules](#)

:::

Motivation

The Cosmos SDK is a framework that makes it easy for developers to build complex decentralized applications from scratch, mainly by composing modules together. As the ecosystem of open-source modules for the Cosmos SDK expands, it will become increasingly likely that some of these modules contain vulnerabilities, as a result of the negligence or malice of their developer.

The Cosmos SDK adopts an [object-capabilities-based approach](#) to help developers better protect their application from unwanted inter-module interactions, and `keeper`s are at the core of this approach. A `keeper` can be considered quite literally to be the gatekeeper of a module's store(s). Each store (typically an [IAVL Store](#)) defined within a module comes with a `storeKey`, which grants unlimited access to it. The module's `keeper` holds this `storeKey` (which should otherwise remain unexposed), and defines [methods](#) for reading and writing to the store(s).

The core idea behind the object-capabilities approach is to only reveal what is necessary to get the work done. In practice, this means that instead of handling permissions of modules through access-control lists, module `keeper`s are passed a reference to the specific instance of the other modules' `keeper`s that they need to access (this is done in the [application's constructor function](#)). As a consequence, a module can only interact with the subset of state defined in another module via the methods exposed by the

instance of the other module's `keeper`. This is a great way for developers to control the interactions that their own module can have with modules developed by external developers.

Type Definition

`keeper`s are generally implemented in a `/keeper/keeper.go` file located in the module's folder. By convention, the type `keeper` of a module is simply named `Keeper` and usually follows the following structure:

```
type Keeper struct {
    // External keepers, if any

    // Store key(s)

    // codec

    // authority
}
```

For example, here is the type definition of the `keeper` from the `staking` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/staking/keeper/keeper.go#L23-L31
```

Let us go through the different parameters:

- An expected `keeper` is a `keeper` external to a module that is required by the internal `keeper` of said module. External `keeper`s are listed in the internal `keeper`'s type definition as interfaces. These interfaces are themselves defined in an `expected_keepers.go` file in the root of the module's folder. In this context, interfaces are used to reduce the number of dependencies, as well as to facilitate the maintenance of the module itself.
- `storeKey`s grant access to the store(s) of the [multistore](#) managed by the module. They should always remain unexposed to external modules.
- `cdc` is the [codec](#) used to marshall and unmarshall structs to/from `[]byte`. The `cdc` can be any of `codec.BinaryCodec`, `codec.JSONCodec` or `codec.Codec` based on your requirements. It can be either a proto or amino codec as long as they implement these interfaces.
- The authority listed is a module account or user account that has the right to change module level parameters. Previously this was handled by the param module, which has been deprecated.

Of course, it is possible to define different types of internal `keeper`s for the same module (e.g. a read-only `keeper`). Each type of `keeper` comes with its own constructor function, which is called from the [application's constructor function](#). This is where `keeper`s are instantiated, and where developers make sure to pass correct instances of modules' `keeper`s to other modules that require them.

Implementing Methods

`Keeper`s primarily expose getter and setter methods for the store(s) managed by their module. These methods should remain as simple as possible and strictly be limited to getting or setting the requested

value, as validity checks should have already been performed by the [Msg server](#) when `keeper`'s methods are called.

Typically, a *getter* method will have the following signature

```
func (k Keeper) Get(ctx sdk.Context, key string) returnType
```

and the method will go through the following steps:

1. Retrieve the appropriate store from the `ctx` using the `storeKey`. This is done through the `KVStore(storeKey sdk.StoreKey)` method of the `ctx`. Then it's preferred to use the `prefix.Store` to access only the desired limited subset of the store for convenience and safety.
2. If it exists, get the `[]byte` value stored at location `[]byte(key)` using the `Get(key []byte)` method of the store.
3. Unmarshal the retrieved value from `[]byte` to `returnType` using the codec `cdc`. Return the value.

Similarly, a *setter* method will have the following signature

```
func (k Keeper) Set(ctx sdk.Context, key string, value valueType)
```

and the method will go through the following steps:

1. Retrieve the appropriate store from the `ctx` using the `storeKey`. This is done through the `KVStore(storeKey sdk.StoreKey)` method of the `ctx`. It's preferred to use the `prefix.Store` to access only the desired limited subset of the store for convenience and safety.
2. Marshal `value` to `[]byte` using the codec `cdc`.
3. Set the encoded value in the store at location `key` using the `Set(key []byte, value []byte)` method of the store.

For more, see an example of `keeper`'s [methods implementation from the staking module](#).

The [module KVStore](#) also provides an `Iterator()` method which returns an `Iterator` object to iterate over a domain of keys.

This is an example from the `auth` module to iterate accounts:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/keeper/account.go
```

sidebar_position: 1

Invariants

:::note Synopsis An invariant is a property of the application that should always be true. In the context of the Cosmos SDK, an `Invariant` is a function that checks for a particular invariant. These functions are useful to detect bugs early on and act upon them to limit their potential consequences (e.g. by halting the chain). They are also useful in the development process of the application to detect bugs via simulations. :::

:::note

Pre-requisite Readings

- [Keepers](#)

...

Implementing Invariants

An `Invariant` is a function that checks for a particular invariant within a module. Module `Invariant`s must follow the `Invariant` type:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/invariant.go#L9
```

The `string` return value is the invariant message, which can be used when printing logs, and the `bool` return value is the actual result of the invariant check.

In practice, each module implements `Invariant`s in a `keeper/invariants.go` file within the module's folder. The standard is to implement one `Invariant` function per logical grouping of invariants with the following model:

```
// Example for an Invariant that checks balance-related invariants

func BalanceInvariants(k Keeper) sdk.Invariant {
    return func(ctx sdk.Context) (string, bool) {
        // Implement checks for balance-related invariants
    }
}
```

Additionally, module developers should generally implement an `AllInvariants` function that runs all the `Invariant`s functions of the module:

```
// AllInvariants runs all invariants of the module.
// In this example, the module implements two Invariants: BalanceInvariants and
// DepositsInvariant

func AllInvariants(k Keeper) sdk.Invariant {

    return func(ctx sdk.Context) (string, bool) {
        res, stop := BalanceInvariants(k)(ctx)
        if stop {
            return res, stop
        }

        return DepositsInvariant(k)(ctx)
    }
}
```

Finally, module developers need to implement the `RegisterInvariants` method as part of the [AppModule interface](#). Indeed, the `RegisterInvariants` method of the module, implemented in the `module/module.go` file, typically only defers the call to a `RegisterInvariants` method implemented in

the `keeper/invariants.go` file. The `RegisterInvariants` method registers a route for each `Invariant` function in the [InvariantRegistry](#):

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/staking/keeper/invariants.go#L12-L22
```

For more, see an example of [Invariant's implementation from the staking module](#).

Invariant Registry

The `InvariantRegistry` is a registry where the `Invariant`s of all the modules of an application are registered. There is only one `InvariantRegistry` per **application**, meaning module developers need not implement their own `InvariantRegistry` when building a module. **All module developers need to do is to register their modules' invariants in the InvariantRegistry, as explained in the section above.** The rest of this section gives more information on the `InvariantRegistry` itself, and does not contain anything directly relevant to module developers.

At its core, the `InvariantRegistry` is defined in the Cosmos SDK as an interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/invariant.go#L14-L17
```

Typically, this interface is implemented in the `keeper` of a specific module. The most used implementation of an `InvariantRegistry` can be found in the `crisis` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/crisis/keeper/keeper.go#L48-L50
```

The `InvariantRegistry` is therefore typically instantiated by instantiating the `keeper` of the `crisis` module in the [application's constructor function](#).

`Invariant`s can be checked manually via [message s](#), but most often they are checked automatically at the end of each block. Here is an example from the `crisis` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/crisis/abci.go#L13-L23
```

In both cases, if one of the `Invariant`s returns false, the `InvariantRegistry` can trigger special logic (e.g. have the application panic and print the `Invariant`'s message in the log).

sidebar_position: 1

Module Genesis

:::note Synopsis Modules generally handle a subset of the state and, as such, they need to define the related subset of the genesis file as well as methods to initialize, verify and export it. :::

:::note

Pre-requisite Readings

- [Module Manager](#)
- [Keepers](#)

:::

Type Definition

The subset of the genesis state defined from a given module is generally defined in a `genesis.proto` file ([more info](#) on how to define protobuf messages). The struct defining the module's subset of the genesis state is usually called `GenesisState` and contains all the module-related values that need to be initialized during the genesis process.

See an example of `GenesisState` protobuf message definition from the `auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/auth/v1beta1/genesis.proto
```

Next we present the main genesis-related methods that need to be implemented by module developers in order for their module to be used in Cosmos SDK applications.

DefaultGenesis

The `DefaultGenesis()` method is a simple method that calls the constructor function for `GenesisState` with the default value for each parameter. See an example from the `auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/module.go#L63-L67
```

ValidateGenesis

The `ValidateGenesis(data GenesisState)` method is called to verify that the provided `genesisState` is correct. It should perform validity checks on each of the parameters listed in `GenesisState`. See an example from the `auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/types/genesis.go#L62-L75
```

Other Genesis Methods

Other than the methods related directly to `GenesisState`, module developers are expected to implement two other methods as part of the [AppModuleGenesis interface](#) (only if the module needs to initialize a subset of state in genesis). These methods are `InitGenesis` and `ExportGenesis`.

InitGenesis

The `InitGenesis` method is executed during `InitChain` when the application is first started. Given a `GenesisState`, it initializes the subset of the state managed by the module by using the module's

[keeper](#) setter function on each parameter within the `GenesisState`.

The [module manager](#) of the application is responsible for calling the `InitGenesis` method of each of the application's modules in order. This order is set by the application developer via the manager's `SetOrderGenesisMethod`, which is called in the [application's constructor function](#).

See an example of `InitGenesis` from the `auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/auth/keeper/genesis.go#L8-L35
```

ExportGenesis

The `ExportGenesis` method is executed whenever an export of the state is made. It takes the latest known version of the subset of the state managed by the module and creates a new `GenesisState` out of it. This is mainly used when the chain needs to be upgraded via a hard fork.

See an example of `ExportGenesis` from the `auth` module.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/auth/keeper/genesis.go#L37-L49
```

GenesisTxHandler

`GenesisTxHandler` is a way for modules to submit state transitions prior to the first block. This is used by `x/genutil` to submit the genesis transactions for the validators to be added to staking.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/core/genesis/txhandler.go#L3-L6
```

sidebar_position: 1

Module Interfaces

:::note Synopsis This document details how to build CLI and REST interfaces for a module. Examples from various Cosmos SDK modules are included. :::

:::note

Pre-requisite Readings

- [Building Modules Intro](#)

:::

CLI

One of the main interfaces for an application is the [command-line interface](#). This entrypoint adds commands from the application's modules enabling end-users to create [messages](#) wrapped in transactions and [queries](#). The CLI files are typically found in the module's `./client/cli` folder.

Transaction Commands

In order to create messages that trigger state changes, end-users must create [transactions](#) that wrap and deliver the messages. A transaction command creates a transaction that includes one or more messages.

Transaction commands typically have their own `tx.go` file that lives within the module's `./client/cli` folder. The commands are specified in getter functions and the name of the function should include the name of the command.

Here is an example from the `x/bank` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/bank/client/cli/tx.go#L37-L76
```

In the example, `NewSendTxCmd()` creates and returns the transaction command for a transaction that wraps and delivers `MsgSend`. `MsgSend` is the message used to send tokens from one account to another.

In general, the getter function does the following:

- **Constructs the command:** Read the [Cobra Documentation](#) for more detailed information on how to create commands.
 - **Use:** Specifies the format of the user input required to invoke the command. In the example above, `send` is the name of the transaction command and `[from_key_or_address]`, `[to_address]`, and `[amount]` are the arguments.
 - **Args:** The number of arguments the user provides. In this case, there are exactly three: `[from_key_or_address]`, `[to_address]`, and `[amount]`.
 - **Short and Long:** Descriptions for the command. A `Short` description is expected. A `Long` description can be used to provide additional information that is displayed when a user adds the `--help` flag.
 - **RunE:** Defines a function that can return an error. This is the function that is called when the command is executed. This function encapsulates all of the logic to create a new transaction.
 - The function typically starts by getting the `clientCtx`, which can be done with `client.GetClientTxContext(cmd)`. The `clientCtx` contains information relevant to transaction handling, including information about the user. In this example, the `clientCtx` is used to retrieve the address of the sender by calling `clientCtx.GetFromAddress()`.
 - If applicable, the command's arguments are parsed. In this example, the arguments `[to_address]` and `[amount]` are both parsed.
 - A [message](#) is created using the parsed arguments and information from the `clientCtx`. The constructor function of the message type is called directly. In this case, `types.NewMsgSend(fromAddr, toAddr, amount)`. Its good practice to call, if possible, the necessary [message validation methods](#) before broadcasting the message.
 - Depending on what the user wants, the transaction is either generated offline or signed and broadcasted to the preconfigured node using `tx.GenerateOrBroadcastTxCLI(clientCtx, flags, msg)`.
- **Adds transaction flags:** All transaction commands must add a set of transaction [flags](#). The transaction flags are used to collect additional information from the user (e.g. the amount of fees

the user is willing to pay). The transaction flags are added to the constructed command using `AddTxFlagsToCmd(cmd)`.

- **Returns the command:** Finally, the transaction command is returned.

Each module can implement `NewTxCmd()`, which aggregates all of the transaction commands of the module. Here is an example from the `x/bank` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/client/cli/tx.go#L20-L35
```

Each module then can also implement a `GetTxCmd()` method that simply returns `NewTxCmd()`. This allows the root command to easily aggregate all of the transaction commands for each module. Here is an example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/module.go#L84-L86
```

Query Commands

:::warning This section is being rewritten. Refer to [AutoCLI](#) while this section is being updated. :::

gRPC

[gRPC](#) is a Remote Procedure Call (RPC) framework. RPC is the preferred way for external clients like wallets and exchanges to interact with a blockchain.

In addition to providing an ABCI query pathway, the Cosmos SDK provides a gRPC proxy server that routes gRPC query requests to ABCI query requests.

In order to do that, modules must implement `RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux)` on `AppModuleBasic` to wire the client gRPC requests to the correct handler inside the module.

Here's an example from the `x/auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/module.go#L71-L76
```

gRPC-gateway REST

Applications need to support web services that use HTTP requests (e.g. a web wallet like [Keplr](#)). [grpc-gateway](#) translates REST calls into gRPC calls, which might be useful for clients that do not use gRPC.

Modules that want to expose REST queries should add `google.api.http` annotations to their `rpc` methods, such as in the example below from the `x/auth` module:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/auth/v1beta1/query.proto#L14-L89
```

gRPC gateway is started in-process along with the application and CometBFT. It can be enabled or disabled by setting gRPC Configuration `enable` in [app.toml](#).

The Cosmos SDK provides a command for generating [Swagger](#) documentation (`protoc-gen-swagger`). Setting `swagger` in [`app.toml`](#) defines if swagger documentation should be automatically registered.

sidebar_position: 1

AutoCLI

:::note Synopsis This document details how to build CLI and REST interfaces for a module. Examples from various Cosmos SDK modules are included. :::

:::note

Pre-requisite Readings

- [Building Modules Intro](#)

:::

The `autocli` package is a [Go library](#) for generating CLI (command line interface) interfaces for Cosmos SDK-based applications. It provides a simple way to add CLI commands to your application by generating them automatically based on your gRPC service definitions. Autocli generates CLI commands and flags directly from your protobuf messages, including options, input parameters, and output parameters. This means that you can easily add a CLI interface to your application without having to manually create and manage commands.

Getting Started

Here are the steps to use the `autocli` package:

1. Define your app's modules that implement the `appmodule.AppModule` interface.
2. Configure how behave `autocli` command generation, by implementing the `func (am AppModule) AutoCLIOptions() *autoclivil.ModuleOptions` method on the module. Learn more [here](#).
3. Use the `autocli.AppOptions` struct to specifies the modules you defined. If you are using the `depinject` package to manage your app's dependencies, it can automatically create an instance of `autocli.AppOptions` based on your app's configuration.
4. Use the `EnhanceRootCommand()` method provided by `autocli` to add the CLI commands for the specified modules to your root command and can also be found in the `client/v2/autocli/app.go` file. Additionally, this method adds the `autocli` functionality to your app's root command. This method is additive only, meaning that it does not create commands if they are already registered for a module. Instead, it adds any missing commands to the root command.

Here's an example of how to use `autocli` :

```

// Define your app's modules
testModules := map[string]appmodule.AppModule{
    "testModule": &TestModule{},
}

// Define the autocli AppOptions
autoCliOpts := autocli.AppOptions{
    Modules: testModules,
}

// Get the root command
rootCmd := &cobra.Command{
    Use: "app",
}

// Enhance the root command with autocli
autocli.EnhanceRootCommand(rootCmd, autoCliOpts)

// Run the root command
if err := rootCmd.Execute(); err != nil {
    fmt.Println(err)
}

```

Flags

`autocli` generates flags for each field in a protobuf message. By default, the names of the flags are generated based on the names of the fields in the message. You can customise the flag names using the `namingOptions` parameter of the `Builder.AddMessageFlags()` method.

To define flags for a message, you can use the `Builder.AddMessageFlags()` method. This method takes the `cobra.Command` instance and the message type as input, and generates flags for each field in the message.

```

https://github.com/cosmos/cosmos-sdk/blob/1ac260cb1c6f05666f47e67f8b2cf6229a55c3b/client/v2/autocli/common.go#L44-L49

```

The `binder` variable returned by the `AddMessageFlags()` method is used to bind the command-line arguments to the fields in the message.

You can also customise the behavior of the flags using the `namingOptions` parameter of the `Builder.AddMessageFlags()` method. This parameter allows you to specify a custom prefix for the flags, and to specify whether to generate flags for repeated fields and whether to generate flags for fields with default values.

Commands and Queries

The `autocli` package generates CLI commands and flags for each method defined in your gRPC service. By default, it generates commands for each RPC method that does not return a stream of messages. The

commands are named based on the name of the service method.

For example, given the following protobuf definition for a service:

```
service MyService {
    rpc MyMethod(MyRequest) returns (MyResponse) {}
}
```

`autocli` will generate a command named `my-method` for the `MyMethod` method. The command will have flags for each field in the `MyRequest` message.

If you want to customise the behavior of a command, you can define a custom command by implementing the `autocli.Command` interface. You can then register the command with the `autocli.Builder` instance for your application.

Similarly, you can define a custom query by implementing the `autocli.Query` interface. You can then register the query with the `autocli.Builder` instance for your application.

To add a custom command or query, you can use the `Builder.AddCustomCommand` or `Builder.AddCustomQuery` methods, respectively. These methods take a `cobra.Command` or `cobra.Command` instance, respectively, which can be used to define the behavior of the command or query.

Advanced Usage

Specifying Subcommands

By default, `autocli` generates a command for each method in your gRPC service. However, you can specify subcommands to group related commands together. To specify subcommands, you can use the `autocliv1.ServiceCommandDescriptor` struct.

This example shows how to use the `autocliv1.ServiceCommandDescriptor` struct to group related commands together and specify subcommands in your gRPC service by defining an instance of `autocliv1.ModuleOptions` in your `autocli.go` file.

```
https://github.com/cosmos/cosmos-
sdk/blob/bcdf81cbaf8d70c4e4fa763f51292d54aed689fd/x/gov/autocli.go#L9-L27
```

The `AutoCLIOptions()` method in the `autocli` package allows you to specify the services and sub-commands to be mapped for your app. In the example code, an instance of the `autocliv1.ModuleOptions` struct is defined in the `appmodule.AppModule` implementation located in the `x/gov/autocli.go` file. This configuration groups related commands together and specifies subcommands for each service.

Positional Arguments

Positional arguments are arguments that are passed to a command without being specified as a flag. They are typically used for providing additional context to a command, such as a filename or search query.

To add positional arguments to a command, you can use the `autocliv1.PositionalArgDescriptor` struct, as seen in the example below. You need to specify the `ProtoField` parameter, which is the name

of the protobuf field that should be used as the positional argument. In addition, if the parameter is a variable-length argument, you can specify the `Varargs` parameter as `true`. This can only be applied to the last positional parameter, and the `ProtoField` must be a repeated field.

Here's an example of how to define a positional argument for the `Account` method of the `auth` service:

```
https://github.com/cosmos/cosmos-
sdk/blob/bcdf81cbaf8d70c4e4fa763f51292d54aed689fd/x/auth/autocli.go#L8-L32
```

Here are some example commands that use the positional arguments we defined above:

To query an account by address:

```
<appd> query auth account cosmos1abcd...xyz
```

To query an account address by account number:

```
<appd> query auth address-by-acc-num 1
```

In both of these commands, the `auth` service is being queried with the `query` subcommand, followed by the specific method being called (`account` or `address-by-acc-num`). The positional argument is included at the end of the command (`cosmos1abcd...xyz` or `1`) to specify the address or account number, respectively.

Customising Flag Names

By default, `autocli` generates flag names based on the names of the fields in your protobuf message. However, you can customise the flag names by providing a `FlagOptions` parameter to the `Builder.AddMessageFlags()` method. This parameter allows you to specify custom names for flags based on the names of the message fields. For example, if you have a message with the fields `test` and `test1`, you can use the following naming options to customise the flags

```
options := autocliv1.RpcCommandOptions{
    FlagOptions: map[string]*autocliv1.FlagOptions{
        "test": { Name: "custom_name", },
        "test1": { Name: "other_name", },
    },
}

builder.AddMessageFlags(message, options)
```

Note that `autocliv1.RpcCommandOptions` is a field of the `autocliv1.ServiceCommandDescriptor` struct, which is defined in the `autocliv1` package. To use this option, you can define an instance of `autocliv1.ModuleOptions` in your `appmodule.AppModule` implementation and specify the `FlagOptions` for the relevant service command descriptor.

Conclusion

`autocli` is a powerful tool for adding CLI interfaces to your Cosmos SDK-based applications. It allows you to easily generate CLI commands and flags from your protobuf messages, and provides many options for customising the behavior of your CLI application.

To further enhance your CLI experience with Cosmos SDK-based blockchains, you can use `Hubl`. `Hubl` is a tool that allows you to query any Cosmos SDK-based blockchain using the new AutoCLI feature of the Cosmos SDK. With hubl, you can easily configure a new chain and query modules with just a few simple commands.

For more information on `Hubl`, including how to configure a new chain and query a module, see the [Hubl documentation](#).

sidebar_position: 1

Recommended Folder Structure

:::note Synopsis This document outlines the recommended structure of Cosmos SDK modules. These ideas are meant to be applied as suggestions. Application developers are encouraged to improve upon and contribute to module structure and development design. :::

Structure

A typical Cosmos SDK module can be structured as follows:

```
proto
└ {project_name}
  └ {module_name}
    └ {proto_version}
      └ {module_name}.proto
      └ event.proto
      └ genesis.proto
      └ query.proto
      └ tx.proto
```

- `{module_name}.proto` : The module's common message type definitions.
- `event.proto` : The module's message type definitions related to events.
- `genesis.proto` : The module's message type definitions related to genesis state.
- `query.proto` : The module's Query service and related message type definitions.
- `tx.proto` : The module's Msg service and related message type definitions.

```
x/{module_name}
├ client
| └ cli
|   └ query.go
|   └ tx.go
|     testutil
|       └ cli_test.go
|       └ suite.go
```

```

├── exported
│   └── exported.go
├── keeper
│   ├── genesis.go
│   ├── grpc_query.go
│   ├── hooks.go
│   ├── invariants.go
│   ├── keeper.go
│   ├── keys.go
│   ├── msg_server.go
│   └── querier.go
├── module
│   └── module.go
│       └── abci.go
│           └── autocli.go
├── simulation
│   ├── decoder.go
│   ├── genesis.go
│   ├── operations.go
│   └── params.go
└── {module_name}.pb.go

```

codec.go
errors.go
events.go
events.pb.go
expected_keepers.go
genesis.go
genesis.pb.go
keys.go
msgs.go
params.go
query.pb.go
tx.pb.go

README.md

- `client/` : The module's CLI client functionality implementation and the module's CLI testing suite.
- `exported/` : The module's exported types - typically interface types. If a module relies on keepers from another module, it is expected to receive the keepers as interface contracts through the `expected_keepers.go` file (see below) in order to avoid a direct dependency on the module implementing the keepers. However, these interface contracts can define methods that operate on and/or return types that are specific to the module that is implementing the keepers and this is where `exported/` comes into play. The interface types that are defined in `exported/` use canonical types, allowing for the module to receive the keepers as interface contracts through the `expected_keepers.go` file. This pattern allows for code to remain DRY and also alleviates import cycle chaos.
- `keeper/` : The module's `Keeper` and `MsgServer` implementation.
- `module/` : The module's `AppModule` and `AppModuleBasic` implementation.
 - `abci.go` : The module's `BeginBlocker` and `EndBlocker` implementations (this file is only required if `BeginBlocker` and/or `EndBlocker` need to be defined).
 - `autocli.go` : The module [autocli](#) options.

- `simulation/` : The module's [simulation](#) package defines functions used by the blockchain simulator application (`simapp`).
 - `README.md` : The module's specification documents outlining important concepts, state storage structure, and message and event type definitions. Learn more how to write module specs in the [spec guidelines](#).
 - The root directory includes type definitions for messages, events, and genesis state, including the type definitions generated by Protocol Buffers.
 - `codec.go` : The module's registry methods for interface types.
 - `errors.go` : The module's sentinel errors.
 - `events.go` : The module's event types and constructors.
 - `expected_keepers.go` : The module's [expected keeper](#) interfaces.
 - `genesis.go` : The module's genesis state methods and helper functions.
 - `keys.go` : The module's store keys and associated helper functions.
 - `msgs.go` : The module's message type definitions and associated methods.
 - `params.go` : The module's parameter type definitions and associated methods.
 - `*.pb.go` : The module's type definitions generated by Protocol Buffers (as defined in the respective `*.proto` files above).
-

sidebar_position: 1

Errors

:::note Synopsis This document outlines the recommended usage and APIs for error handling in Cosmos SDK modules. :::

Modules are encouraged to define and register their own errors to provide better context on failed message or handler execution. Typically, these errors should be common or general errors which can be further wrapped to provide additional specific execution context.

Registration

Modules should define and register their custom errors in `x/{module}/errors.go`. Registration of errors is handled via the [errors package](#).

Example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/distribution/types/errors.go
```

Each custom module error must provide the codespace, which is typically the module name (e.g. "distribution") and is unique per module, and a uint32 code. Together, the codespace and code provide a globally unique Cosmos SDK error. Typically, the code is monotonically increasing but does not necessarily have to be. The only restrictions on error codes are the following:

- Must be greater than one, as a code value of one is reserved for internal errors.
- Must be unique within the module.

Note, the Cosmos SDK provides a core set of *common* errors. These errors are defined in [types/errors/errors.go](#).

Wrapping

The custom module errors can be returned as their concrete type as they already fulfill the `error` interface. However, module errors can be wrapped to provide further context and meaning to failed execution.

Example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/keeper/keeper.go#L141-L182
```

Regardless if an error is wrapped or not, the Cosmos SDK's `errors` package provides a function to determine if an error is of a particular kind via `Is`.

ABCI

If a module error is registered, the Cosmos SDK `errors` package allows ABCI information to be extracted through the `ABCIInfo` function. The package also provides `ResponseCheckTx` and `ResponseDeliverTx` as auxiliary functions to automatically get `CheckTx` and `DeliverTx` responses from an error.

sidebar_position: 1

Upgrading Modules

:::note Synopsis [In-Place Store Migrations](#) allow your modules to upgrade to new versions that include breaking changes. This document outlines how to build modules to take advantage of this functionality. :::

:::note

Pre-requisite Readings

- [In-Place Store Migration](#)

:::

Consensus Version

Successful upgrades of existing modules require each `AppModule` to implement the function `ConsensusVersion() uint64`.

- The versions must be hard-coded by the module developer.
- The initial version **must** be set to 1.

Consensus versions serve as state-breaking versions of app modules and must be incremented when the module introduces breaking changes.

Registering Migrations

To register the functionality that takes place during a module upgrade, you must register which migrations you want to take place.

Migration registration takes place in the `Configurator` using the `RegisterMigration` method. The `AppModule` reference to the configurator is in the `RegisterServices` method.

You can register one or more migrations. If you register more than one migration script, list the migrations in increasing order and ensure there are enough migrations that lead to the desired consensus version. For example, to migrate to version 3 of a module, register separate migrations for version 1 and version 2 as shown in the following example:

```
func (am AppModule) RegisterServices(cfg module.Configurator) {
    // --snip--
    cfg.RegisterMigration(types.ModuleName, 1, func(ctx sdk.Context) error {
        // Perform in-place store migrations from ConsensusVersion 1 to 2.
    })
    cfg.RegisterMigration(types.ModuleName, 2, func(ctx sdk.Context) error {
        // Perform in-place store migrations from ConsensusVersion 2 to 3.
    })
}
```

Since these migrations are functions that need access to a Keeper's store, use a wrapper around the keepers called `Migrator` as shown in this example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/keeper/migrations.go
```

Writing Migration Scripts

To define the functionality that takes place during an upgrade, write a migration script and place the functions in a `migrations/` directory. For example, to write migration scripts for the bank module, place the functions in `x/bank/migrations/`. Use the recommended naming convention for these functions. For example, `v2bank` is the script that migrates the package `x/bank/migrations/v2`:

```
// Migrating bank module from version 1 to 2
func (m Migrator) MigrateTo2(ctx sdk.Context) error {
    return v2bank.MigrateStore(ctx, m.keeper.storeKey) // v2bank is package
    `x/bank/migrations/v2`.
}
```

To see example code of changes that were implemented in a migration of balance keys, check out [migrateBalanceKeys](#). For context, this code introduced migrations of the bank store that updated addresses to be prefixed by their length in bytes as outlined in [ADR-028](#).

sidebar_position: 1

Module Simulation

:::note

Pre-requisite Readings

- [Cosmos Blockchain Simulator](#) :::

Synopsis

This document details how to define each module simulation functions to be integrated with the application `SimulationManager`.

- [Simulation package](#)
 - [Store decoders](#)
 - [Randomized genesis](#)
 - [Randomized parameter changes](#)
 - [Random weighted operations](#)
 - [Random proposal contents](#)
- [Registering simulation functions](#)
- [App Simulator manager](#)

Simulation package

Every module that implements the Cosmos SDK simulator needs to have a `x/<module>/simulation` package which contains the primary functions required by the fuzz tests: store decoders, randomized genesis state and parameters, weighted operations and proposal contents.

Store decoders

Registering the store decoders is required for the `AppImportExport`. This allows for the key-value pairs from the stores to be decoded (*i.e* unmarshalled) to their corresponding types. In particular, it matches the key to a concrete type and then unmarshals the value from the `KVPair` to the type provided.

You can use the example [here](#) from the distribution module to implement your store decoders.

Randomized genesis

The simulator tests different scenarios and values for genesis parameters in order to fully test the edge cases of specific modules. The `simulator` package from each module must expose a `RandomizedGenState` function to generate the initial random `GenesisState` from a given seed.

Once the module genesis parameter are generated randomly (or with the key and values defined in a `params` file), they are marshaled to JSON format and added to the app genesis JSON to use it on the simulations.

You can check an example on how to create the randomized genesis [here](#).

Randomized parameter changes

The simulator is able to test parameter changes at random. The simulator package from each module must contain a `RandomizedParams` func that will simulate parameter changes of the module throughout the simulations lifespan.

You can see how an example of what is needed to fully test parameter changes [here](#)

Random weighted operations

Operations are one of the crucial parts of the Cosmos SDK simulation. They are the transactions (`Msg`) that are simulated with random field values. The sender of the operation is also assigned randomly.

Operations on the simulation are simulated using the full [transaction cycle](#) of a `ABCI` application that exposes the `BaseApp`.

Shown below is how weights are set:

```
https://github.com/cosmos/cosmos-sdk/blob/v/x/staking/simulation/operations.go#L19-L86
```

As you can see, the weights are predefined in this case. Options exist to override this behavior with different weights. One option is to use `*rand.Rand` to define a random weight for the operation, or you can inject your own predefined weights.

Here is how one can override the above package `simappparams`.

```
https://github.com/cosmos/cosmos-sdk/blob/v/Makefile#L293-L299
```

For the last test a tool called [runsim](#) is used, this is used to parallelize go test instances, provide info to Github and slack integrations to provide information to your team on how the simulations are running.

Random proposal contents

Randomized governance proposals are also supported on the Cosmos SDK simulator. Each module must define the governance proposal `Content`s that they expose and register them to be used on the parameters.

Registering simulation functions

Now that all the required functions are defined, we need to integrate them into the module pattern within the `module.go`:

```
https://github.com/cosmos/cosmos-sdk/blob/v/x/distribution/module.go#L180-L203
```

App Simulator manager

The following step is setting up the `SimulatorManager` at the app level. This is required for the simulation test files on the next step.

```
type CustomApp struct {
    ...
}
```

```
    sm *module.SimulationManager  
}
```

Then at the instantiation of the application, we create the `SimulationManager` instance in the same way we create the `ModuleManager` but this time we only pass the modules that implement the simulation functions from the `AppModuleSimulation` interface described above.

```
func NewCustomApp(...) {  
    // create the simulation manager and define the order of the modules for  
    // deterministic simulations  
    app.sm = module.NewSimulationManager(  
        auth.New AppModule(app.accountKeeper),  
        bank.New AppModule(app.bankKeeper, app.accountKeeper),  
        supply.New AppModule(app.supplyKeeper, app.accountKeeper),  
        gov.New AppModule(app.govKeeper, app.accountKeeper, app.supplyKeeper),  
        mint.New AppModule(app.mintKeeper),  
        distr.New AppModule(app.distrKeeper, app.accountKeeper, app.supplyKeeper,  
            app.stakingKeeper),  
        staking.New AppModule(app.stakingKeeper, app.accountKeeper, app.supplyKeeper),  
        slashing.New AppModule(app.slashingKeeper, app.accountKeeper, app.stakingKeeper),  
    )  
  
    // register the store decoders for simulation tests  
    app.sm.RegisterStoreDecoders()  
    ...  
}
```

sidebar_position: 1

Modules depinjected-ready

:::note

Pre-requisite Readings

- [Depinjection Documentation](#)

:::

`depinject` is used to wire any module in `app.go`. All core modules are already configured to support dependency injection.

To work with `depinject` a module must define its configuration and requirements so that `depinject` can provide the right dependencies.

In brief, as a module developer, the following steps are required:

1. Define the module configuration using Protobuf
2. Define the module dependencies in `x/{moduleName}/module.go`

A chain developer can then use the module by following these two steps:

1. Configure the module in `app_config.go` or `app.yaml`
2. Inject the module in `app.go`

Module Configuration

The module available configuration is defined in a Protobuf file, located at
`{moduleName}/module/v1/module.proto`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/proto/cosmos/group/module/v1/module.proto
```

- `go_import` must point to the Go package of the custom module.
- Message fields define the module configuration. That configuration can be set in the `app_config.go` / `app.yaml` file for a chain developer to configure the module.
 Taking `group` as example, a chain developer is able to decide, thanks to `uint64 max_metadata_len`, what the maximum metadata length allowed for a group proposal is.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/simapp/app_config.go#L228-L234
```

That message is generated using `pulsar` (by running `make proto-gen`). In the case of the `group` module, this file is generated here: <https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/api/cosmos/group/module/v1/module.pulsar.go>.

The part that is relevant for the module configuration is:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/api/cosmos/group/module/v1/module.pulsar.go#L515-L527
```

:::note Pulsar is optional. The official [protoc-gen-go](#) can be used as well. :::

Dependency Definition

Once the configuration proto is defined, the module's `module.go` must define what dependencies are required by the module. The boilerplate is similar for all modules.

:::warning All methods, structs and their fields must be public for `depinject`. :::

1. Import the module configuration generated package:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/group/module/module.go#L12-L14
```

Define an `init()` function for defining the `providers` of the module configuration:
 This registers the module configuration message and the wiring of the module.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
```

```
alpha.0/x/group/module/module.go#L194-L199
```

2. Ensure that the module implements the `appmodule.AppModule` interface:

```
https://github.com/cosmos/cosmos-
sdk/blob/v0.47.0/x/group/module/module.go#L58-L64
```

3. Define a struct that inherits `depinj.In` and define the module inputs (i.e. module dependencies):

- o `depinj` provides the right dependencies to the module.
- o `depinj` also checks that all dependencies are provided.

:::tip For making a dependency optional, add the `optional:"true"` struct tag.

:::

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/group/module/module.go#L201-L211
```

4. Define the module outputs with a public struct that inherits `depinj.Out` : The module outputs are the dependencies that the module provides to other modules. It is usually the module itself and its keeper.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/group/module/module.go#L213-L218
```

5. Create a function named `ProvideModule` (as called in 1.) and use the inputs for instantiating the module outputs.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/group/module/module.go#L220-L235
```

The `ProvideModule` function should return an instance of `cosmossdk.io/core/appmodule.AppModule` which implements one or more app module extension interfaces for initializing the module.

Following is the complete app wiring configuration for `group` :

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/group/module/module.go#L194-L235
```

The module is now ready to be used with `depinj` by a chain developer.

Integrate in an application

The App Wiring is done in `app_config.go` / `app.yaml` and `app_v2.go` and is explained in detail in the [overview of `app_v2.go`](#).

sidebar_position: 1

Testing

The Cosmos SDK contains different types of [tests](#). These tests have different goals and are used at different stages of the development cycle. We advice, as a general rule, to use tests at all stages of the development cycle. It is advised, as a chain developer, to test your application and modules in a similar way than the SDK.

The rationale behind testing can be found in [ADR-59](#).

Unit Tests

Unit tests are the lowest test category of the [test pyramid](#). All packages and modules should have unit test coverage. Modules should have their dependencies mocked: this means mocking keepers.

The SDK uses `mockgen` to generate mocks for keepers:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/scripts/mockgen.sh#L3-L6
```

You can read more about mockgen [here](#).

Example

As an example, we will walkthrough the [keeper tests](#) of the `x/gov` module.

The `x/gov` module has a `Keeper` type, which requires a few external dependencies (ie. imports outside `x/gov` to work properly).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/gov/keeper/keeper.go#L22-L24
```

In order to only test `x/gov`, we mock the [expected keepers](#) and instantiate the `Keeper` with the mocked dependencies. Note that we may need to configure the mocked dependencies to return the expected values:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/gov/keeper/common_test.go#L67-L81
```

This allows us to test the `x/gov` module without having to import other modules.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/gov/keeper/keeper_test.go#L3-L42
```

We can then create unit tests using the newly created `Keeper` instance.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/gov/keeper/keeper_test.go#L83-L107
```

Integration Tests

Integration tests are at the second level of the [test pyramid](#). In the SDK, we locate our integration tests under [/tests/integrations](#).

The goal of these integration tests is to test how a component interacts with other dependencies. Compared to unit tests, integration tests do not mock dependencies. Instead, they use the direct dependencies of the component. This differs as well from end-to-end tests, which test the component with a full application.

Integration tests interact with the tested module via the defined `Msg` and `Query` services. The result of the test can be verified by checking the state of the application, by checking the emitted events or the response. It is advised to combine two of these methods to verify the result of the test.

The SDK provides small helpers for quickly setting up an integration tests. These helpers can be found at <https://github.com/cosmos/cosmos-sdk/blob/main/testutil/integration>.

Example

```
https://github.com/cosmos/cosmos-
sdk/blob/a2f73a7dd37bea0ab303792c55fa1e4e1db3b898/testutil/integration/example_test.go
L116
```

Deterministic and Regression tests

Tests are written for queries in the Cosmos SDK which have `module_query_safe` Protobuf annotation.

Each query is tested using 2 methods:

- Use property-based testing with the [rapid](#) library. The property that is tested is that the query response and gas consumption are the same upon 1000 query calls.
- Regression tests are written with hardcoded responses and gas, and verify they don't change upon 1000 calls and between SDK patch versions.

Here's an example of regression tests:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/tests/integration/bank/keeper/deterministic_test.go#L134-L151
```

Simulations

Simulations uses as well a minimal application, built with [depinject](#):

:::note You can as well use the `AppConfig configurator` for creating an `AppConfig inline`. There is no difference between those two ways, use whichever you prefer. :::

Following is an example for `x/gov/ simulations`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/gov/simulation/operations_test.go#L406-L430
```

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/gov/simulation/operations\_test.go#L90-L132
```

End-to-end Tests

End-to-end tests are at the top of the [test pyramid](#). They must test the whole application flow, from the user perspective (for instance, CLI tests). They are located under [/tests/e2e](#).

For that, the SDK is using `simapp` but you should use your own application (`appd`). Here are some examples:

- SDK E2E tests: <https://github.com/cosmos/cosmos-sdk/tree/main/tests/e2e>.
- Cosmos Hub E2E tests: <https://github.com/cosmos/gaia/tree/main/tests/e2e>.
- Osmosis E2E tests: <https://github.com/osmosis-labs/osmosis/tree/main/tests/e2e>.

:::note warning The SDK is in the process of creating its E2E tests, as defined in [ADR-59](#). This page will eventually be updated with better examples. :::

Learn More

Learn more about testing scope in [ADR-59](#).

sidebar_position: 1

BaseApp

:::note Synopsis This document describes `BaseApp`, the abstraction that implements the core functionalities of a Cosmos SDK application. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK application](#)
- [Lifecycle of a Cosmos SDK transaction](#)

:::

Introduction

`BaseApp` is a base type that implements the core of a Cosmos SDK application, namely:

- The [Application Blockchain Interface](#), for the state-machine to communicate with the underlying consensus engine (e.g. CometBFT).
- [Service Routers](#), to route messages and queries to the appropriate module.
- Different [states](#), as the state-machine can have different volatile states updated based on the ABCI message received.

The goal of `BaseApp` is to provide the fundamental layer of a Cosmos SDK application that developers can easily extend to build their own custom application. Usually, developers will create a custom type for their application, like so:

```

type App struct {
    // reference to a BaseApp
    *baseapp.BaseApp

    // list of application store keys

    // list of application keepers

    // module manager
}

```

Extending the application with `BaseApp` gives the former access to all of `BaseApp`'s methods. This allows developers to compose their custom application with the modules they want, while not having to concern themselves with the hard work of implementing the ABCI, the service routers and state management logic.

Type Definition

The `BaseApp` type holds many important parameters for any Cosmos SDK based application.

<https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/baseapp.go#L58-L182>

Let us go through the most important components.

Note: Not all parameters are described, only the most important ones. Refer to the type definition for the full list.

First, the important parameters that are initialized during the bootstrapping of the application:

- [CommitMultiStore](#): This is the main store of the application, which holds the canonical state that is committed at the [end of each block](#). This store is **not** cached, meaning it is not used to update the application's volatile (un-committed) states. The `CommitMultiStore` is a multi-store, meaning a store of stores. Each module of the application uses one or multiple `KVStores` in the multi-store to persist their subset of the state.
- Database: The `db` is used by the `CommitMultiStore` to handle data persistence.
- [Msg Service Router](#): The `msgServiceRouter` facilitates the routing of `sdk.Msg` requests to the appropriate module `Msg` service for processing. Here a `sdk.Msg` refers to the transaction component that needs to be processed by a service in order to update the application state, and not to ABCI message which implements the interface between the application and the underlying consensus engine.
- [gRPC Query Router](#): The `grpcQueryRouter` facilitates the routing of gRPC queries to the appropriate module for it to be processed. These queries are not ABCI messages themselves, but they are relayed to the relevant module's gRPC `Query` service.
- [TxDecoder](#): It is used to decode raw transaction bytes relayed by the underlying CometBFT engine.
- [AnteHandler](#): This handler is used to handle signature verification, fee payment, and other pre-message execution checks when a transaction is received. It's executed during `CheckTx/RecheckTx` and `FinalizeBlock`.

- `InitChainer`, `BeginBlocker` and `EndBlocker`: These are the functions executed when the application receives the `InitChain` and `FinalizeBlock` ABCI messages from the underlying CometBFT engine.

Then, parameters used to define [volatile states](#) (i.e. cached states):

- `checkState` : This state is updated during `CheckTx`, and reset on `Commit`.
- `finalizeBlockState` : This state is updated during `FinalizeBlock`, and set to `nil` on `Commit` and gets re-initialized on `FinalizeBlock`.
- `processProposalState` : This state is updated during `ProcessProposal`.
- `prepareProposalState` : This state is updated during `PrepareProposal`.
- `voteExtensionState` : This state is updated during `ExtendVote` & `VerifyVoteExtension`.

Finally, a few more important parameters:

- `voteInfos` : This parameter carries the list of validators whose precommit is missing, either because they did not vote or because the proposer did not include their vote. This information is carried by the [Context](#) and can be used by the application for various things like punishing absent validators.
- `minGasPrices` : This parameter defines the minimum gas prices accepted by the node. This is a **local** parameter, meaning each full-node can set a different `minGasPrices`. It is used in the `AnteHandler` during `CheckTx`, mainly as a spam protection mechanism. The transaction enters the [mempool](#) only if the gas prices of the transaction are greater than one of the minimum gas price in `minGasPrices` (e.g. if `minGasPrices == 1uatom,1photon`, the `gas-price` of the transaction must be greater than `1uatom OR 1photon`).
- `appVersion` : Version of the application. It is set in the [application's constructor function](#).

Constructor

```
func NewBaseApp(
    name string, logger log.Logger, db dbm.DB, txDecoder sdk.TxDecoder, options
    ...func(*BaseApp),
) *BaseApp {
    // ...
}
```

The `BaseApp` constructor function is pretty straightforward. The only thing worth noting is the possibility to provide additional [options](#) to the `BaseApp`, which will execute them in order. The `options` are generally setter functions for important parameters, like `SetPruning()` to set pruning options or `SetMinGasPrices()` to set the node's min-gas-prices.

Naturally, developers can add additional `options` based on their application's needs.

State Updates

The `BaseApp` maintains four primary volatile states and a root or main state. The main state is the canonical state of the application and the volatile states, `checkState`, `prepareProposalState`, `processProposalState`, `voteExtensionState` and `finalizeBlockState` are used to handle state transitions in-between the main state made during `Commit`.

Internally, there is only a single `CommitMultiStore` which we refer to as the main or root state. From this root state, we derive four volatile states by using a mechanism called *store branching* (performed by `CacheWrap` function). The types can be illustrated as follows:



InitChain State Updates

During `InitChain`, the four volatile states, `checkState`, `prepareProposalState`, `processProposalState` and `finalizeBlockState` are set by branching the root `CommitMultiStore`. Any subsequent reads and writes happen on branched versions of the `CommitMultiStore`. To avoid unnecessary roundtrip to the main state, all reads to the branched store are cached.



CheckTx State Updates

During `CheckTx`, the `checkState`, which is based off of the last committed state from the root store, is used for any reads and writes. Here we only execute the `AnteHandler` and verify a service router exists for every message in the transaction. Note, when we execute the `AnteHandler`, we branch the already branched `checkState`. This has the side effect that if the `AnteHandler` fails, the state transitions won't be reflected in the `checkState` -- i.e. `checkState` is only updated on success.



PrepareProposal State Updates

During `PrepareProposal`, the `prepareProposalState` is set by branching the root `CommitMultiStore`. The `prepareProposalState` is used for any reads and writes that occur during the `PrepareProposal` phase. The function uses the `Select()` method of the mempool to iterate over the transactions. `runTx` is then called, which encodes and validates each transaction and from there the `AnteHandler` is executed. If successful, valid transactions are returned inclusive of the events, tags, and data generated during the execution of the proposal. The described behavior is that of the default handler, applications have the flexibility to define their own [custom mempool handlers](#).



ProcessProposal State Updates

During `ProcessProposal`, the `processProposalState` is set based off of the last committed state from the root store and is used to process a signed proposal received from a validator. In this state, `runTx` is called and the `AnteHandler` is executed and the context used in this state is built with information from the header and the main state, including the minimum gas prices, which are also set. Again we want to highlight that the described behavior is that of the default handler and applications have the flexibility to define their own [custom mempool handlers](#).



FinalizeBlock State Updates

During `FinalizeBlock`, the `finalizeBlockState` is set for use during transaction execution and `endblock`. The `finalizeBlockState` is based off of the last committed state from the root store and is branched. Note, the `finalizeBlockState` is set to `nil` on [Commit](#).

The state flow for transaction execution is nearly identical to `CheckTx` except state transitions occur on the `finalizeBlockState` and messages in a transaction are executed. Similarly to `CheckTx`, state transitions occur on a doubly branched state -- `finalizeBlockState`. Successful message execution results in writes being committed to `finalizeBlockState`. Note, if message execution fails, state transitions from the AnteHandler are persisted.

Commit State Updates

During `Commit` all the state transitions that occurred in the `finalizeBlockState` are finally written to the root `CommitMultiStore` which in turn is committed to disk and results in a new application root hash. These state transitions are now considered final. Finally, the `checkState` is set to the newly committed state and `finalizeBlockState` is set to `nil` to be reset on `FinalizeBlock`.



ParamStore

During `InitChain`, the `RequestInitChain` provides `ConsensusParams` which contains parameters related to block execution such as maximum gas and size in addition to evidence parameters. If these parameters are non-nil, they are set in the BaseApp's `ParamStore`. Behind the scenes, the `ParamStore` is managed by an `x/consensus_params` module. This allows the parameters to be tweaked via on-chain governance.

Service Routers

When messages and queries are received by the application, they must be routed to the appropriate module in order to be processed. Routing is done via `BaseApp`, which holds a `msgServiceRouter` for messages, and a `grpcQueryRouter` for queries.

Msg Service Router

`sdk.Msg`s need to be routed after they are extracted from transactions, which are sent from the underlying CometBFT engine via the [CheckTx](#) and [FinalizeBlock](#) ABCI messages. To do so, `BaseApp` holds a `msgServiceRouter` which maps fully-qualified service methods (`string`, defined in each module's Protobuf `Msg` service) to the appropriate module's `MsgServer` implementation.

The [default msgServiceRouter included in BaseApp](#) is stateless. However, some applications may want to make use of more stateful routing mechanisms such as allowing governance to disable certain routes or point them to new modules for upgrade purposes. For this reason, the `sdk.Context` is also passed into

each [route handler inside `msgServiceRouter`](#). For a stateless router that doesn't want to make use of this, you can just ignore the `ctx`.

The application's `msgServiceRouter` is initialized with all the routes using the application's [module manager](#) (via the `RegisterServices` method), which itself is initialized with all the application's modules in the application's [constructor](#).

gRPC Query Router

Similar to `sdk.Msg`s, [queries](#) need to be routed to the appropriate module's [Query service](#). To do so, `BaseApp` holds a `grpcQueryRouter`, which maps modules' fully-qualified service methods (`string`, defined in their Protobuf `Query gRPC`) to their `QueryServer` implementation. The `grpcQueryRouter` is called during the initial stages of query processing, which can be either by directly sending a gRPC query to the gRPC endpoint, or via the [Query ABCI message](#) on the CometBFT RPC endpoint.

Just like the `msgServiceRouter`, the `grpcQueryRouter` is initialized with all the query routes using the application's [module manager](#) (via the `RegisterServices` method), which itself is initialized with all the application's modules in the application's [constructor](#).

Main ABCI 2.0 Messages

The [Application-Blockchain Interface](#) (ABCI) is a generic interface that connects a state-machine with a consensus engine to form a functional full-node. It can be wrapped in any language, and needs to be implemented by each application-specific blockchain built on top of an ABCI-compatible consensus engine like CometBFT.

The consensus engine handles two main tasks:

- The networking logic, which mainly consists in gossiping block parts, transactions and consensus votes.
- The consensus logic, which results in the deterministic ordering of transactions in the form of blocks.

It is **not** the role of the consensus engine to define the state or the validity of transactions. Generally, transactions are handled by the consensus engine in the form of `[]bytes`, and relayed to the application via the ABCI to be decoded and processed. At key moments in the networking and consensus processes (e.g. beginning of a block, commit of a block, reception of an unconfirmed transaction, ...), the consensus engine emits ABCI messages for the state-machine to act on.

Developers building on top of the Cosmos SDK need not implement the ABCI themselves, as `BaseApp` comes with a built-in implementation of the interface. Let us go through the main ABCI messages that `BaseApp` implements:

- [Prepare_Proposal](#)
- [Process_Proposal](#)
- [CheckTx](#)
- [FinalizeBlock](#)
- [ExtendVote](#)
- [VerifyVoteExtension](#)

Prepare Proposal

The `PrepareProposal` function is part of the new methods introduced in Application Blockchain Interface (ABCI++) in CometBFT and is an important part of the application's overall governance system. In the Cosmos SDK, it allows the application to have more fine-grained control over the transactions that are processed, and ensures that only valid transactions are committed to the blockchain.

Here is how the `PrepareProposal` function can be implemented:

1. Extract the `sdk.Msg`'s from the transaction.
2. Perform *stateful* checks by calling `Validate()` on each of the `sdk.Msg`'s. This is done after *stateless* checks as *stateful* checks are more computationally expensive. If `Validate()` fails, `PrepareProposal` returns before running further checks, which saves resources.
3. Perform any additional checks that are specific to the application, such as checking account balances, or ensuring that certain conditions are met before a transaction is proposed. These are processed by the consensus engine, if necessary.
4. Return the updated transactions to be processed by the consensus engine

Note that, unlike `CheckTx()`, `PrepareProposal` processes `sdk.Msg`'s, so it can directly update the state. However, unlike `FinalizeBlock()`, it does not commit the state updates. It's important to exercise caution when using `PrepareProposal` as incorrect coding could affect the overall liveness of the network.

It's important to note that `PrepareProposal` complements the `ProcessProposal` method which is executed after this method. The combination of these two methods means that it is possible to guarantee that no invalid transactions are ever committed. Furthermore, such a setup can give rise to other interesting use cases such as Oracles, threshold decryption and more.

`PrepareProposal` returns a response to the underlying consensus engine of type [`abci.ResponseCheckTx`](#). The response contains:

- `Code (uint32)` : Response Code. 0 if successful.
- `Data ([]byte)` : Result bytes, if any.
- `Log (string)` : The output of the application's logger. May be non-deterministic.
- `Info (string)` : Additional information. May be non-deterministic.

Process Proposal

The `ProcessProposal` function is called by the BaseApp as part of the ABCI message flow, and is executed during the `FinalizeBlock` phase of the consensus process. The purpose of this function is to give more control to the application for block validation, allowing it to check all transactions in a proposed block before the validator sends the prevote for the block. It allows a validator to perform application-dependent work in a proposed block, enabling features such as immediate block execution, and allows the Application to reject invalid blocks.

The `ProcessProposal` function performs several key tasks, including:

1. Validating the proposed block by checking all transactions in it.
2. Checking the proposed block against the current state of the application, to ensure that it is valid and that it can be executed.
3. Updating the application's state based on the proposal, if it is valid and passes all checks.
4. Returning a response to CometBFT indicating the result of the proposal processing.

The `ProcessProposal` is an important part of the application's overall governance system. It is used to manage the network's parameters and other key aspects of its operation. It also ensures that the coherence property is adhered to i.e. all honest validators must accept a proposal by an honest proposer.

It's important to note that `ProcessProposal` complements the `PrepareProposal` method which enables the application to have more fine-grained transaction control by allowing it to reorder, drop, delay, modify, and even add transactions as they see necessary. The combination of these two methods means that it is possible to guarantee that no invalid transactions are ever committed. Furthermore, such a setup can give rise to other interesting use cases such as Oracles, threshold decryption and more.

CometBFT calls it when it receives a proposal and the CometBFT algorithm has not locked on a value. The Application cannot modify the proposal at this point but can reject it if it is invalid. If that is the case, CometBFT will prevote `nil` on the proposal, which has strong liveness implications for CometBFT. As a general rule, the Application SHOULD accept a prepared proposal passed via `ProcessProposal`, even if a part of the proposal is invalid (e.g., an invalid transaction); the Application can ignore the invalid part of the prepared proposal at block execution time.

However, developers must exercise greater caution when using these methods. Incorrectly coding these methods could affect liveness as CometBFT is unable to receive 2/3 valid precommits to finalize a block.

`ProcessProposal` returns a response to the underlying consensus engine of type [`abci.ResponseCheckTx`](#). The response contains:

- `Code (uint32)` : Response Code. `0` if successful.
- `Data ([]byte)` : Result bytes, if any.
- `Log (string)` : The output of the application's logger. May be non-deterministic.
- `Info (string)` : Additional information. May be non-deterministic.

CheckTx

`CheckTx` is sent by the underlying consensus engine when a new unconfirmed (i.e. not yet included in a valid block) transaction is received by a full-node. The role of `CheckTx` is to guard the full-node's mempool (where unconfirmed transactions are stored until they are included in a block) from spam transactions. Unconfirmed transactions are relayed to peers only if they pass `CheckTx`.

`CheckTx()` can perform both *stateful* and *stateless* checks, but developers should strive to make the checks **lightweight** because gas fees are not charged for the resources (CPU, data load...) used during the `CheckTx`.

In the Cosmos SDK, after [decoding transactions](#), `CheckTx()` is implemented to do the following checks:

1. Extract the `sdk.Msg`s from the transaction.
2. **Optionally** perform *stateless* checks by calling `ValidateBasic()` on each of the `sdk.Msg`s.
This is done first, as *stateless* checks are less computationally expensive than *stateful* checks. If `ValidateBasic()` fail, `CheckTx` returns before running *stateful* checks, which saves resources.
This check is still performed for messages that have not yet migrated to the new message validation mechanism defined in [RFC 001](#) and still have a `ValidateBasic()` method.
3. Perform non-module related *stateful* checks on the [account](#). This step is mainly about checking that the `sdk.Msg` signatures are valid, that enough fees are provided and that the sending account has enough funds to pay for said fees. Note that no precise `gas` counting occurs here, as `sdk.Msg`s are not processed. Usually, the [AnteHandler](#) will check that the `gas` provided with the transaction is superior to a minimum reference gas amount based on the raw transaction size, in order to avoid spam with transactions that provide 0 gas.

`CheckTx` does **not** process `sdk.Msg`s - they only need to be processed when the canonical state need to be updated, which happens during `FinalizeBlock`.

Steps 2. and 3. are performed by the `AnteHandler` in the `RunTx()` function, which `CheckTx()` calls with the `runTxModeCheck` mode. During each step of `CheckTx()`, a special [volatile state](#) called `checkState` is updated. This state is used to keep track of the temporary changes triggered by the `CheckTx()` calls of each transaction without modifying the [main canonical state](#). For example, when a transaction goes through `CheckTx()`, the transaction's fees are deducted from the sender's account in `checkState`. If a second transaction is received from the same account before the first is processed, and the account has consumed all its funds in `checkState` during the first transaction, the second transaction will fail `CheckTx()` and be rejected. In any case, the sender's account will not actually pay the fees until the transaction is actually included in a block, because `checkState` never gets committed to the main state. The `checkState` is reset to the latest state of the main state each time a block gets [committed](#).

`CheckTx` returns a response to the underlying consensus engine of type `abci.ResponseCheckTx`. The response contains:

- `Code (uint32) : Response Code. 0 if successful.`
- `Data ([]byte) : Result bytes, if any.`
- `Log (string) : The output of the application's logger. May be non-deterministic.`
- `Info (string) : Additional information. May be non-deterministic.`
- `GasWanted (int64) : Amount of gas requested for transaction. It is provided by users when they generate the transaction.`
- `GasUsed (int64) : Amount of gas consumed by transaction. During CheckTx, this value is computed by multiplying the standard cost of a transaction byte by the size of the raw transaction.`

Next is an example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/ante/basic.go#L102
```

- `Events ([]cmn.KVPair) : Key-Value tags for filtering and indexing transactions (eg. by account).`
See [event s](#) for more.
- `Codespace (string) : Namespace for the Code.`

RecheckTx

After `Commit`, `CheckTx` is run again on all transactions that remain in the node's local mempool excluding the transactions that are included in the block. To prevent the mempool from rechecking all transactions every time a block is committed, the configuration option `mempool.recheck=false` can be set. As of Tendermint v0.32.1, an additional `Type` parameter is made available to the `CheckTx` function that indicates whether an incoming transaction is new (`CheckTxType_New`), or a recheck (`CheckTxType_Recheck`). This allows certain checks like signature verification can be skipped during `CheckTxType_Recheck`.

RunTx, AnteHandler, RunMsgs, PostHandler

RunTx

`RunTx` is called from `CheckTx / Finalizeblock` to handle the transaction, with `execModeCheck` or `execModeFinalize` as parameter to differentiate between the two modes of execution. Note that when `RunTx` receives a transaction, it has already been decoded.

The first thing `RunTx` does upon being called is to retrieve the `context`'s `CacheMultiStore` by calling the `getContextForTx()` function with the appropriate mode (either `runTxModeCheck` or

`execModeFinalize`). This `CacheMultiStore` is a branch of the main store, with cache functionality (for query requests), instantiated during `FinalizeBlock` for transaction execution and during the `Commit` of the previous block for `CheckTx`. After that, two `defer func()` are called for `gas` management. They are executed when `runTx` returns and make sure `gas` is actually consumed, and will throw errors, if any.

After that, `RunTx()` calls `ValidateBasic()`, when available and for backward compatibility, on each `sdk.Msg` in the `Tx`, which runs preliminary *stateless* validity checks. If any `sdk.Msg` fails to pass `ValidateBasic()`, `RunTx()` returns with an error.

Then, the `anteHandler` of the application is run (if it exists). In preparation of this step, both the `checkState / finalizeBlockState`'s `context` and `context`'s `CacheMultiStore` are branched using the `cacheTxContext()` function.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/baseapp.go#L663-L680
```

This allows `RunTx` not to commit the changes made to the state during the execution of `anteHandler` if it ends up failing. It also prevents the module implementing the `anteHandler` from writing to state, which is an important part of the [object-capabilities](#) of the Cosmos SDK.

Finally, the `RunMsgs()` function is called to process the `sdk.Msg`s in the `Tx`. In preparation of this step, just like with the `anteHandler`, both the `checkState / finalizeBlockState`'s `context` and `context`'s `CacheMultiStore` are branched using the `cacheTxContext()` function.

AnteHandler

The `AnteHandler` is a special handler that implements the `AnteHandler` interface and is used to authenticate the transaction before the transaction's internal messages are processed.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/handler.go#L6-L18
```

The `AnteHandler` is theoretically optional, but still a very important component of public blockchain networks. It serves 3 primary purposes:

- Be a primary line of defense against spam and second line of defense (the first one being the mempool) against transaction replay with fees deduction and [sequence](#) checking.
- Perform preliminary *stateful* validity checks like ensuring signatures are valid or that the sender has enough funds to pay for fees.
- Play a role in the incentivisation of stakeholders via the collection of transaction fees.

`BaseApp` holds an `anteHandler` as parameter that is initialized in the [application's constructor](#). The most widely used `anteHandler` is the [auth module](#).

Click [here](#) for more on the `anteHandler`.

RunMsgs

`RunMsgs` is called from `RunTx` with `runTxModeCheck` as parameter to check the existence of a route for each message the transaction, and with `execModeFinalize` to actually process the `sdk.Msg`s.

First, it retrieves the `sdk.Msg`'s fully-qualified type name, by checking the `type_url` of the Protobuf `Any` representing the `sdk.Msg`. Then, using the application's `msgServiceRouter`, it checks for the existence of `Msg` service method related to that `type_url`. At this point, if `mode == runTxModeCheck`, `RunMsgs` returns. Otherwise, if `mode == execModeFinalize`, the `Msg service` RPC is executed, before `RunMsgs` returns.

PostHandler

`PostHandler` is similar to `AnteHandler`, but it, as the name suggests, executes custom post tx processing logic after `RunMsgs` is called. `PostHandler` receives the `Result` of the the `RunMsgs` in order to enable this customizable behavior.

Like `AnteHandler`s, `PostHandler`s are theoretically optional, one use case for `PostHandler`s is transaction tips (enabled by default in simapp). Other use cases like unused gas refund can also be enabled by `PostHandler`s.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/posthandler/post.go#L1-L15
```

Note, when `PostHandler`s fail, the state from `runMsgs` is also reverted, effectively making the transaction fail.

Other ABCI Messages

InitChain

The [InitChain ABCI message](#) is sent from the underlying CometBFT engine when the chain is first started. It is mainly used to **initialize** parameters and state like:

- [Consensus Parameters](#) via `setConsensusParams`.
- [checkState and finalizeBlockState](#) via `setState`.
- The [block gas meter](#), with infinite gas to process genesis transactions.

Finally, the `InitChain(req abci.RequestInitChain)` method of `BaseApp` calls the [initChainer\(\)](#) of the application in order to initialize the main state of the application from the `genesis` file and, if defined, call the [InitGenesis](#) function of each of the application's modules.

FinalizeBlock

The [FinalizeBlock ABCI message](#) is sent from the underlying CometBFT engine when a block proposal created by the correct proposer is received. The previous `BeginBlock`, `DeliverTx` and `Endblock` calls are private methods on the `BaseApp` struct.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci.go#L623
```

BeginBlock

- Initialize [finalizeBlockState](#) with the latest header using the `req abci.RequestFinalizeBlock` passed as parameter via the `setState` function.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/baseapp.go#L682-L706
```

This function also resets the [main gas meter](#).

- Initialize the [block gas meter](#) with the `maxGas` limit. The `gas` consumed within the block cannot go above `maxGas`. This parameter is defined in the application's consensus parameters.
- Run the application's [`beginBlocker\(\)`](#), which mainly runs the [`BeginBlocker\(\)`](#) method of each of the application's modules.
- Set the [`VoteInfos`](#) of the application, i.e. the list of validators whose *precommit* for the previous block was included by the proposer of the current block. This information is carried into the [`Context`](#) so that it can be used during transaction execution and `EndBlock`.

Transaction Execution

When the underlying consensus engine receives a block proposal, each transaction in the block needs to be processed by the application. To that end, the underlying consensus engine sends the transactions in `FinalizeBlock` message to the application for each transaction in a sequential order.

Before the first transaction of a given block is processed, a [volatile state](#) called `finalizeBlockState` is initialized during `FinalizeBlock`. This state is updated each time a transaction is processed via `FinalizeBlock`, and committed to the [main state](#) when the block is [committed](#), after what it is set to `nil`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/baseapp.go#L708-L743
```

Transaction execution within `FinalizeBlock` performs the **exact same steps as `CheckTx`**, with a little caveat at step 3 and the addition of a fifth step:

1. The `AnteHandler` does **not** check that the transaction's `gas-prices` is sufficient. That is because the `min-gas-prices` value `gas-prices` is checked against is local to the node, and therefore what is enough for one full-node might not be for another. This means that the proposer can potentially include transactions for free, although they are not incentivised to do so, as they earn a bonus on the total fee of the block they propose.
2. For each `sdk.Msg` in the transaction, route to the appropriate module's Protobuf [`Msg service`](#). Additional *stateful* checks are performed, and the branched multistore held in `finalizeBlockState`'s `context` is updated by the module's `keeper`. If the `Msg` service returns successfully, the branched multistore held in `context` is written to `finalizeBlockState CacheMultiStore`.

During the additional fifth step outlined in (2), each read/write to the store increases the value of `GasConsumed`. You can find the default cost of each operation:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/gas.go#L230-L241
```

At any point, if `GasConsumed > GasWanted`, the function returns with `Code != 0` and the execution fails.

Each transaction returns a response to the underlying consensus engine of type [`abci.ExecTxResult`](#).

The response contains:

- `Code (uint32) : Response Code. 0 if successful.`
 - `Data ([]byte) : Result bytes, if any.`
 - `Log (string) : The output of the application's logger. May be non-deterministic.`
 - `Info (string) : Additional information. May be non-deterministic.`
 - `GasWanted (int64) : Amount of gas requested for transaction. It is provided by users when they generate the transaction.`
 - `GasUsed (int64) : Amount of gas consumed by transaction. During transaction execution, this value is computed by multiplying the standard cost of a transaction byte by the size of the raw transaction, and by adding gas each time a read/write to the store occurs.`
 - `Events ([]cmn.KVPair) : Key-Value tags for filtering and indexing transactions (eg. by account).`
- See [`event s`](#) for more.
- `Codespace (string) : Namespace for the Code.`

EndBlock

`EndBlock` is run after transaction execution completes. It allows developers to have logic be executed at the end of each block. In the Cosmos SDK, the bulk `EndBlock()` method is to run the application's `EndBlocker()`, which mainly runs the `EndBlocker()` method of each of the application's modules.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/baseapp.go#L747-L769
```

Commit

The [`Commit ABCI message`](#) is sent from the underlying CometBFT engine after the full-node has received *precommits* from 2/3+ of validators (weighted by voting power). On the `BaseApp` end, the `Commit(res abci.ResponseCommit)` function is implemented to commit all the valid state transitions that occurred during `FinalizeBlock` and to reset state for the next block.

To commit state-transitions, the `Commit` function calls the `Write()` function on `finalizeBlockState.ms`, where `finalizeBlockState.ms` is a branched multistore of the main store `app.cms`. Then, the `Commit` function sets `checkState` to the latest header (obtained from `finalizeBlockState.ctx.BlockHeader`) and `finalizeBlockState` to `nil`.

Finally, `Commit` returns the hash of the commitment of `app.cms` back to the underlying consensus engine. This hash is used as a reference in the header of the next block.

Info

The [`Info ABCI message`](#) is a simple query from the underlying consensus engine, notably used to sync the latter with the application during a handshake that happens on startup. When called, the `Info(res abci.ResponseInfo)` function from `BaseApp` will return the application's name, version and the hash of the last commit of `app.cms`.

Query

The [`Query ABCI message`](#) is used to serve queries received from the underlying consensus engine, including queries received via RPC like CometBFT RPC. It used to be the main entrypoint to build interfaces

with the application, but with the introduction of [gRPC queries](#) in Cosmos SDK v0.40, its usage is more limited. The application must respect a few rules when implementing the `Query` method, which are outlined [here](#).

Each CometBFT `query` comes with a `path`, which is a `string` which denotes what to query. If the `path` matches a gRPC fully-qualified service method, then `BaseApp` will defer the query to the `grpcQueryRouter` and let it handle it like explained [above](#). Otherwise, the `path` represents a query that is not (yet) handled by the gRPC router. `BaseApp` splits the `path` string with the `/` delimiter. By convention, the first element of the split string (`split[0]`) contains the category of `query` (`app`, `p2p`, `store` or `custom`). The `BaseApp` implementation of the `Query(req abci.RequestQuery)` method is a simple dispatcher serving these 4 main categories of queries:

- Application-related queries like querying the application's version, which are served via the `handleQueryApp` method.
- Direct queries to the multistore, which are served by the `handlerQueryStore` method. These direct queries are different from custom queries which go through `app.queryRouter`, and are mainly used by third-party service provider like block explorers.
- P2P queries, which are served via the `handleQueryP2P` method. These queries return either `app.addrPeerFilter` or `app.ipPeerFilter` that contain the list of peers filtered by address or IP respectively. These lists are first initialized via `options` in `BaseApp`'s [constructor](#).

ExtendVote

`ExtendVote` allows an application to extend a pre-commit vote with arbitrary data. This process does NOT have to be deterministic and the data returned can be unique to the validator process.

In the Cosmos-SDK this is implemented as a NoOp:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci_utils.go#L274-L281
```

VerifyVoteExtension

`VerifyVoteExtension` allows an application to verify that the data returned by `ExtendVote` is valid. This process does NOT have to be deterministic and the data returned can be unique to the validator process.

In the Cosmos-SDK this is implemented as a NoOp:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/abci_utils.go#L282-L288
```

sidebar_position: 1

Transactions

:::note Synopsis `Transactions` are objects created by end-users to trigger state changes in the application. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK Application](#)

⋮

Transactions

Transactions are comprised of metadata held in [contexts](#) and [sdk.Msg](#)s that trigger state changes within a module through the module's Protobuf [Msg service](#).

When users want to interact with an application and make state changes (e.g. sending coins), they create transactions. Each of a transaction's `sdk.Msg` must be signed using the private key associated with the appropriate account(s), before the transaction is broadcasted to the network. A transaction must then be included in a block, validated, and approved by the network through the consensus process. To read more about the lifecycle of a transaction, click [here](#).

Type Definition

Transaction objects are Cosmos SDK types that implement the `Tx` interface

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/tx\_msg.go#L51-L56
```

It contains the following methods:

- **GetMsgs:** unwraps the transaction and returns a list of contained `sdk.Msg`s - one transaction may have one or multiple messages, which are defined by module developers.
- **ValidateBasic:** lightweight, [stateless](#) checks used by ABCI messages [CheckTx](#) and [DeliverTx](#) to make sure transactions are not invalid. For example, the `auth` module's `ValidateBasic` function checks that its transactions are signed by the correct number of signers and that the fees do not exceed what the user's maximum. When `runTx` is checking a transaction created from the `auth` module, it first runs `ValidateBasic` on each message, then runs the `auth` module `AnteHandler` which calls `ValidateBasic` for the transaction itself.

:::note This function is different from the deprecated `sdk.Msg` [ValidateBasic](#) methods, which was performing basic validity checks on messages only. :::

As a developer, you should rarely manipulate `Tx` directly, as `Tx` is really an intermediate type used for transaction generation. Instead, developers should prefer the `TxBuilder` interface, which you can learn more about [below](#).

Signing Transactions

Every message in a transaction must be signed by the addresses specified by its `GetSigners`. The Cosmos SDK currently allows signing transactions in two different ways.

`SIGN_MODE_DIRECT` (preferred)

The most used implementation of the `Tx` interface is the Protobuf `Tx` message, which is used in `SIGN_MODE_DIRECT`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L13-L26
```

Because Protobuf serialization is not deterministic, the Cosmos SDK uses an additional `TxRaw` type to denote the pinned bytes over which a transaction is signed. Any user can generate a valid `body` and `auth_info` for a transaction, and serialize these two messages using Protobuf. `TxRaw` then pins the user's exact binary representation of `body` and `auth_info`, called respectively `body_bytes` and `auth_info_bytes`. The document that is signed by all signers of the transaction is `SignDoc` (deterministically serialized using [ADR-027](#)):

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L48-L65
```

Once signed by all signers, the `body_bytes`, `auth_info_bytes` and `signatures` are gathered into `TxRaw`, whose serialized bytes are broadcasted over the network.

SIGN_MODE_LEGACY_AMINO_JSON

The legacy implementation of the `Tx` interface is the `StdTx` struct from `x/auth`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/migrations/legacytx/stdtx.go#L83-L90
```

The document signed by all signers is `StdSignDoc`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/migrations/legacytx/stdsign.go#L31-L45
```

which is encoded into bytes using Amino JSON. Once all signatures are gathered into `StdTx`, `StdTx` is serialized using Amino JSON, and these bytes are broadcasted over the network.

Other Sign Modes

The Cosmos SDK also provides a couple of other sign modes for particular use cases.

SIGN_MODE_DIRECT_AUX

`SIGN_MODE_DIRECT_AUX` is a sign mode released in the Cosmos SDK v0.46 which targets transactions with multiple signers. Whereas `SIGN_MODE_DIRECT` expects each signer to sign over both `TxBody` and `AuthInfo` (which includes all other signers' signer infos, i.e. their account sequence, public key and mode info), `SIGN_MODE_DIRECT_AUX` allows N-1 signers to only sign over `TxBody` and *their own* signer info. Moreover, each auxiliary signer (i.e. a signer using `SIGN_MODE_DIRECT_AUX`) doesn't need to sign over the fees:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L67-L98
```

The use case is a multi-signer transaction, where one of the signers is appointed to gather all signatures, broadcast the signature and pay for fees, and the others only care about the transaction body. This

generally allows for a better multi-signing UX. If Alice, Bob and Charlie are part of a 3-signer transaction, then Alice and Bob can both use `SIGN_MODE_DIRECT_AUX` to sign over the `TxBody` and their own signer info (no need an additional step to gather other signers' ones, like in `SIGN_MODE_DIRECT`), without specifying a fee in their SignDoc. Charlie can then gather both signatures from Alice and Bob, and create the final transaction by appending a fee. Note that the fee payer of the transaction (in our case Charlie) must sign over the fees, so must use `SIGN_MODE_DIRECT` or `SIGN_MODE_LEGACY_AMINO_JSON`.

A concrete use case is implemented in [transaction tips](#): the tipper may use `SIGN_MODE_DIRECT_AUX` to specify a tip in the transaction, without signing over the actual transaction fees. Then, the fee payer appends fees inside the tipper's desired `TxBody`, and as an exchange for paying the fees and broadcasting the transaction, receives the tipper's transaction tips as payment.

`SIGN_MODE_TEXTUAL`

`SIGN_MODE_TEXTUAL` is a new sign mode for delivering a better signing experience on hardware wallets, it is currently still under implementation. If you wish to learn more, please refer to [ADR-050](#).

Custom Sign modes

There is the opportunity to add your own custom sign mode to the Cosmos-SDK. While we can not accept the implementation of the sign mode to the repository, we can accept a pull request to add the custom signmode to the `SignMode` enum located [here](#).

Transaction Process

The process of an end-user sending a transaction is:

- decide on the messages to put into the transaction,
- generate the transaction using the Cosmos SDK's `TxBuilder`,
- broadcast the transaction using one of the available interfaces.

The next paragraphs will describe each of these components, in this order.

Messages

:::tip Module `sdk.Msg` s are not to be confused with [ABCI Messages](#) which define interactions between the CometBFT and application layers. :::

Messages (or `sdk.Msg` s) are module-specific objects that trigger state transitions within the scope of the module they belong to. Module developers define the messages for their module by adding methods to the Protobuf [Msg service](#), and also implement the corresponding `MsgServer`.

Each `sdk.Msg` s is related to exactly one Protobuf [Msg service](#) RPC, defined inside each module's `tx.proto` file. A SDK app router automatically maps every `sdk.Msg` to a corresponding RPC. Protobuf generates a `MsgServer` interface for each module `Msg` service, and the module developer needs to implement this interface. This design puts more responsibility on module developers, allowing application developers to reuse common functionalities without having to implement state transition logic repetitively.

To learn more about Protobuf `Msg` services and how to implement `MsgServer`, click [here](#).

While messages contain the information for state transition logic, a transaction's other metadata and relevant information are stored in the `TxBuilder` and `Context`.

Transaction Generation

The `TxBuilder` interface contains data closely related with the generation of transactions, which an end-user can freely set to generate the desired transaction:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/tx_config.go#L40-L53
```

- `Msg` s, the array of `messages` included in the transaction.
- `GasLimit`, option chosen by the users for how to calculate how much gas they will need to pay.
- `Memo`, a note or comment to send with the transaction.
- `FeeAmount`, the maximum amount the user is willing to pay in fees.
- `TimeoutHeight`, block height until which the transaction is valid.
- `Signatures`, the array of signatures from all signers of the transaction.

As there are currently two sign modes for signing transactions, there are also two implementations of `TxBuilder`:

- `wrapper` for creating transactions for `SIGN_MODE_DIRECT`,
- `StdTxBuilder` for `SIGN_MODE_LEGACY_AMINO_JSON`.

However, the two implementation of `TxBuilder` should be hidden away from end-users, as they should prefer using the overarching `TxConfig` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/tx_config.go#L24-L34
```

`TxConfig` is an app-wide configuration for managing transactions. Most importantly, it holds the information about whether to sign each transaction with `SIGN_MODE_DIRECT` or `SIGN_MODE_LEGACY_AMINO_JSON`. By calling `txBuilder := txConfig.NewTxBuilder()`, a new `TxBuilder` will be created with the appropriate sign mode.

Once `TxBuilder` is correctly populated with the setters exposed above, `TxConfig` will also take care of correctly encoding the bytes (again, either using `SIGN_MODE_DIRECT` or `SIGN_MODE_LEGACY_AMINO_JSON`). Here's a pseudo-code snippet of how to generate and encode a transaction, using the `TxEncoder()` method:

```
txBuilder := txConfig.NewTxBuilder()
txBuilder.SetMsgs(...) // and other setters on txBuilder

bz, err := txConfig.TxE encoder()(txBuilder.GetTx())
// bz are bytes to be broadcasted over the network
```

Broadcasting the Transaction

Once the transaction bytes are generated, there are currently three ways of broadcasting it.

CLI

Application developers create entry points to the application by creating a [command-line interface, gRPC and/or REST interface](#), typically found in the application's `./cmd` folder. These interfaces allow users to interact with the application through command-line.

For the [command-line interface](#), module developers create subcommands to add as children to the application top-level transaction command `TxCmd`. CLI commands actually bundle all the steps of transaction processing into one simple command: creating messages, generating transactions and broadcasting. For concrete examples, see the [Interacting with a Node](#) section. An example transaction made using CLI looks like:

```
simd tx send $MY_VALIDATOR_ADDRESS $RECIPIENT 1000stake
```

gRPC

[gRPC](#) is the main component for the Cosmos SDK's RPC layer. Its principal usage is in the context of modules' [Query services](#). However, the Cosmos SDK also exposes a few other module-agnostic gRPC services, one of them being the `Tx` service:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/service.proto
```

The `Tx` service exposes a handful of utility functions, such as simulating a transaction or querying a transaction, and also one method to broadcast transactions.

Examples of broadcasting and simulating a transaction are shown [here](#).

REST

Each gRPC method has its corresponding REST endpoint, generated using [gRPC-gateway](#). Therefore, instead of using gRPC, you can also use HTTP to broadcast the same transaction, on the `POST /cosmos/tx/v1beta1/txs` endpoint.

An example can be seen [here](#)

CometBFT RPC

The three methods presented above are actually higher abstractions over the CometBFT RPC `/broadcast_tx_{async, sync, commit}` endpoints, documented [here](#). This means that you can use the CometBFT RPC endpoints directly to broadcast the transaction, if you wish so.

sidebar_position: 1

Context

:::note Synopsis The `context` is a data structure intended to be passed from function to function that carries information about the current state of the application. It provides access to a branched storage (a

safe branch of the entire state) as well as useful objects and information like `gasMeter`, `block height`, `consensus parameters` and more. :::

:::note

Pre-requisites Readings

- [Anatomy of a Cosmos SDK Application](#)
- [Lifecycle of a Transaction](#)

:::

Context Definition

The Cosmos SDK `Context` is a custom data structure that contains Go's stdlib [context](#) as its base, and has many additional types within its definition that are specific to the Cosmos SDK. The `Context` is integral to transaction processing in that it allows modules to easily access their respective `store` in the [multistore](#) and retrieve transactional context such as the block header and gas meter.

<https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/context.go#L41-L67>

- **Base Context:** The base type is a Go [Context](#), which is explained further in the [Go Context Package](#) section below.
- **Multistore:** Every application's `BaseApp` contains a [CommitMultistore](#) which is provided when a `Context` is created. Calling the `KVStore()` and `TransientStore()` methods allows modules to fetch their respective [KVStore](#) using their unique `StoreKey`.
- **Header:** The [header](#) is a Blockchain type. It carries important information about the state of the blockchain, such as block height and proposer of the current block.
- **Header Hash:** The current block header hash, obtained during `abci.FinalizeBlock`.
- **Chain ID:** The unique identification number of the blockchain a block pertains to.
- **Transaction Bytes:** The `[]byte` representation of a transaction being processed using the context. Every transaction is processed by various parts of the Cosmos SDK and consensus engine (e.g. CometBFT) throughout its [lifecycle](#), some of which do not have any understanding of transaction types. Thus, transactions are marshaled into the generic `[]byte` type using some kind of [encoding format](#) such as [Amino](#).
- **Logger:** A `logger` from the CometBFT libraries. Learn more about logs [here](#). Modules call this method to create their own unique module-specific logger.
- **VoteInfo:** A list of the ABCI type [VoteInfo](#), which includes the name of a validator and a boolean indicating whether they have signed the block.
- **Gas Meters:** Specifically, a `gasMeter` for the transaction currently being processed using the context and a `blockGasMeter` for the entire block it belongs to. Users specify how much in fees they wish to pay for the execution of their transaction; these gas meters keep track of how much [gas](#) has been used in the transaction or block so far. If the gas meter runs out, execution halts.
- **CheckTx Mode:** A boolean value indicating whether a transaction should be processed in `CheckTx` or `DeliverTx` mode.
- **Min Gas Price:** The minimum [gas](#) price a node is willing to take in order to include a transaction in its block. This price is a local value configured by each node individually, and should therefore **not be used in any functions used in sequences leading to state-transitions**.
- **Consensus Params:** The ABCI type [Consensus Parameters](#), which specify certain limits for the blockchain, such as maximum gas for a block.

- **Event Manager:** The event manager allows any caller with access to a `Context` to emit `Events`. Modules may define module specific `Events` by defining various `Types` and `Attributes` or use the common definitions found in `types/`. Clients can subscribe or query for these `Events`. These `Events` are collected throughout `FinalizeBlock` and are returned to CometBFT for indexing.
- **Priority:** The transaction priority, only relevant in `CheckTx`.
- **KV GasConfig :** Enables applications to set a custom `GasConfig` for the `KVStore`.
- **Transient KV GasConfig :** Enables applications to set a custom `GasConfig` for the transient `KVStore`.

Go Context Package

A basic `Context` is defined in the [Golang Context Package](#). A `Context` is an immutable data structure that carries request-scoped data across APIs and processes. Contexts are also designed to enable concurrency and to be used in goroutines.

Contexts are intended to be **immutable**; they should never be edited. Instead, the convention is to create a child context from its parent using a `with` function. For example:

```
childCtx = parentCtx.WithBlockHeader(header)
```

The [Golang Context Package](#) documentation instructs developers to explicitly pass a context `ctx` as the first argument of a process.

Store branching

The `Context` contains a `MultiStore`, which allows for branching and caching functionality using `CacheMultiStore` (queries in `CacheMultiStore` are cached to avoid future round trips). Each `KVStore` is branched in a safe and isolated ephemeral storage. Processes are free to write changes to the `CacheMultiStore`. If a state-transition sequence is performed without issue, the store branch can be committed to the underlying store at the end of the sequence or disregard them if something goes wrong. The pattern of usage for a Context is as follows:

1. A process receives a Context `ctx` from its parent process, which provides information needed to perform the process.
2. The `ctx.ms` is a **branched store**, i.e. a branch of the [multistore](#) is made so that the process can make changes to the state as it executes, without changing the original `ctx.ms`. This is useful to protect the underlying multistore in case the changes need to be reverted at some point in the execution.
3. The process may read and write from `ctx` as it is executing. It may call a subprocess and pass `ctx` to it as needed.
4. When a subprocess returns, it checks if the result is a success or failure. If a failure, nothing needs to be done - the branch `ctx` is simply discarded. If successful, the changes made to the `CacheMultiStore` can be committed to the original `ctx.ms` via `Write()`.

For example, here is a snippet from the `runTx` function in [baseapp](#):

```
runMsgCtx, msCache := app.cacheTxContext(ctx, txBytes)
result = app.runMsgs(runMsgCtx, msgs, mode)
```

```

result.GasWanted = gasWanted
if mode != runTxModeDeliver {
    return result
}
if result.IsOK() {
    msCache.Write()
}

```

Here is the process:

1. Prior to calling `runMsgs` on the message(s) in the transaction, it uses `app.cacheTxContext()` to branch and cache the context and multistore.
 2. `runMsgCtx` - the context with branched store, is used in `runMsgs` to return a result.
 3. If the process is running in `checkTxMode`, there is no need to write the changes - the result is returned immediately.
 4. If the process is running in `deliverTxMode` and the result indicates a successful run over all the messages, the branched multistore is written back to the original.
-

sidebar_position: 1

Node Client (Daemon)

:::note Synopsis The main endpoint of a Cosmos SDK application is the daemon client, otherwise known as the full-node client. The full-node runs the state-machine, starting from a genesis file. It connects to peers running the same client in order to receive and relay transactions, block proposals and signatures. The full-node is constituted of the application, defined with the Cosmos SDK, and of a consensus engine connected to the application via the ABCI. :::

:::note

Pre-requisite Readings

- [Anatomy of an SDK application](#)

:::

main function

The full-node client of any Cosmos SDK application is built by running a `main` function. The client is generally named by appending the `-d` suffix to the application name (e.g. `appd` for an application named `app`), and the `main` function is defined in a `./appd/cmd/main.go` file. Running this function creates an executable `appd` that comes with a set of commands. For an app named `app`, the main command is `appd start`, which starts the full-node.

In general, developers will implement the `main.go` function with the following structure:

- First, an `encodingCodec` is instantiated for the application.
- Then, the `config` is retrieved and config parameters are set. This mainly involves setting the Bech32 prefixes for [addresses](#).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/config.go#L14-L29
```

- Using [cobra](#), the root command of the full-node client is created. After that, all the custom commands of the application are added using the `AddCommand()` method of `rootCmd`.
- Add default server commands to `rootCmd` using the `server.AddCommands()` method. These commands are separated from the ones added above since they are standard and defined at Cosmos SDK level. They should be shared by all Cosmos SDK-based applications. They include the most important command: the [start command](#).
- Prepare and execute the `executor`.

```
https://github.com/cometbft/cometbft/blob/v0.37.0/libs/cli/setup.go#L74-L78
```

See an example of `main` function from the `simapp` application, the Cosmos SDK's application for demo purposes:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/main.go
```

start command

The `start` command is defined in the `/server` folder of the Cosmos SDK. It is added to the root command of the full-node client in the [main function](#) and called by the end-user to start their node:

```
# For an example app named "app", the following command starts the full-node.  
appd start  
  
# Using the Cosmos SDK's own simapp, the following commands start the simapp node.  
simd start
```

As a reminder, the full-node is composed of three conceptual layers: the networking layer, the consensus layer and the application layer. The first two are generally bundled together in an entity called the consensus engine (CometBFT by default), while the third is the state-machine defined with the help of the Cosmos SDK. Currently, the Cosmos SDK uses CometBFT as the default consensus engine, meaning the `start` command is implemented to boot up a CometBFT node.

The flow of the `start` command is pretty straightforward. First, it retrieves the `config` from the `context` in order to open the `db` (a [leveldb](#) instance by default). This `db` contains the latest known state of the application (empty if the application is started from the first time).

With the `db`, the `start` command creates a new instance of the application using an `appCreator` function:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/start.go#L220
```

Note that an `appCreator` is a function that fulfills the `AppCreator` signature:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/types/app.go#L68
```

In practice, the [constructor of the application](#) is passed as the `appCreator`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L294-L308
```

Then, the instance of `app` is used to instantiate a new CometBFT node:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/start.go#L341-L378
```

The CometBFT node can be created with `app` because the latter satisfies the [abci.Application interface](#) (given that `app` extends [baseapp](#)). As part of the `node.New` method, CometBFT makes sure that the height of the application (i.e. number of blocks since genesis) is equal to the height of the CometBFT node. The difference between these two heights should always be negative or null. If it is strictly negative, `node.New` will replay blocks until the height of the application reaches the height of the CometBFT node. Finally, if the height of the application is `0`, the CometBFT node will call [InitChain](#) on the application to initialize the state from the genesis file.

Once the CometBFT node is instantiated and in sync with the application, the node can be started:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/start.go#L350-L352
```

Upon starting, the node will bootstrap its RPC and P2P server and start dialing peers. During handshake with its peers, if the node realizes they are ahead, it will query all the blocks sequentially in order to catch up. Then, it will wait for new block proposals and block signatures from validators in order to make progress.

Other commands

To discover how to concretely run a node and interact with it, please refer to our [Running a Node, API and CLI](#) guide.

sidebar_position: 1

Store

:::note Synopsis A store is a data structure that holds the state of the application. :::

:::note

Pre-requisite Readings

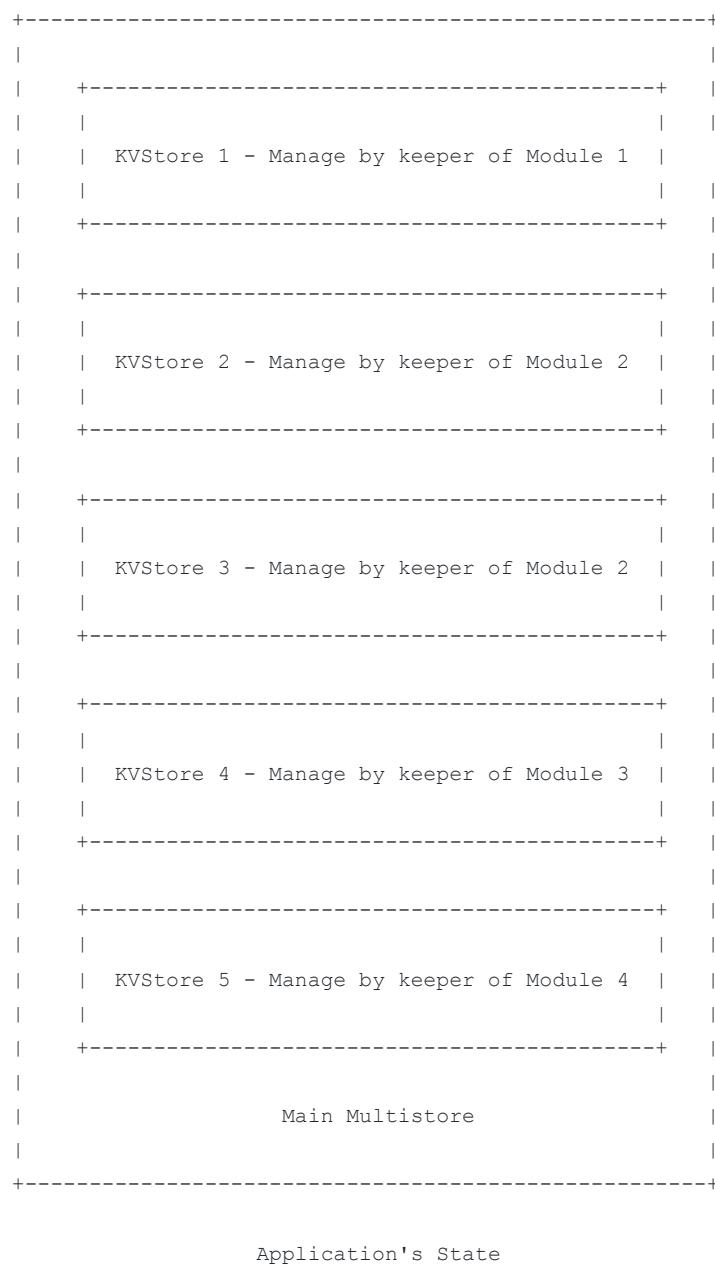
- [Anatomy of a Cosmos SDK application](#)

:::

Introduction to Cosmos SDK Stores

The Cosmos SDK comes with a large set of stores to persist the state of applications. By default, the main store of Cosmos SDK applications is a `multistore`, i.e. a store of stores. Developers can add any number of key-value stores to the multistore, depending on their application needs. The multistore exists to support

the modularity of the Cosmos SDK, as it lets each module declare and manage their own subset of the state. Key-value stores in the multistore can only be accessed with a specific capability `key`, which is typically held in the [`keeper`](#) of the module that declared the store.



Store Interface

At its very core, a Cosmos SDK `store` is an object that holds a `CacheWrapper` and has a `GetStoreType()` method:

<https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L15-L18>

The `GetStoreType` is a simple method that returns the type of store, whereas a `CacheWrapper` is a simple interface that implements store read caching and write branching through `Write` method:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L287-L320
```

Branching and cache is used ubiquitously in the Cosmos SDK and required to be implemented on every store type. A storage branch creates an isolated, ephemeral branch of a store that can be passed around and updated without affecting the main underlying store. This is used to trigger temporary state-transitions that may be reverted later should an error occur. Read more about it in [context](#)

Commit Store

A commit store is a store that has the ability to commit changes made to the underlying tree or db. The Cosmos SDK differentiates simple stores from commit stores by extending the basic store interfaces with a `Committer`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L32-L37
```

The `Committer` is an interface that defines methods to persist changes to disk:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L20-L30
```

The `CommitID` is a deterministic commit of the state tree. Its hash is returned to the underlying consensus engine and stored in the block header. Note that commit store interfaces exist for various purposes, one of which is to make sure not every object can commit the store. As part of the [object-capabilities model](#) of the Cosmos SDK, only `baseapp` should have the ability to commit stores. For example, this is the reason why the `ctx.KVStore()` method by which modules typically access stores returns a `KVStore` and not a `CommitKVStore`.

The Cosmos SDK comes with many types of stores, the most used being [`CommitMultiStore`](#), [`KVStore`](#) and [`GasKv store`](#). [Other types of stores](#) include `Transient` and `TraceKV` stores.

Multistore

Multistore Interface

Each Cosmos SDK application holds a multistore at its root to persist its state. The multistore is a store of `KVStores` that follows the `Multistore` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L123-L155
```

If tracing is enabled, then branching the multistore will firstly wrap all the underlying `KVStore` in [`TraceKv.Store`](#).

CommitMultiStore

The main type of `Multistore` used in the Cosmos SDK is `CommitMultiStore`, which is an extension of the `Multistore` interface:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L164-L227
```

As for concrete implementation, the [`rootMulti.Store`] is the go-to implementation of the `CommitMultiStore` interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/rootmulti/store.go#L53-L77
```

The `rootMulti.Store` is a base-layer multistore built around a `db` on top of which multiple `KVStores` can be mounted, and is the default multistore store used in [baseapp](#).

CacheMultiStore

Whenever the `rootMulti.Store` needs to be branched, a [`cachemulti.Store`](#) is used.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/cachemulti/store.go#L19-L33
```

`cachemulti.Store` branches all substores (creates a virtual store for each substore) in its constructor and hold them in `Store.stores`. Moreover caches all read queries. `Store.GetKVStore()` returns the store from `Store.stores`, and `Store.Write()` recursively calls `CacheWrap.Write()` on all the substores.

Base-layer KVStores

KVStore and CommitKVStore Interfaces

A `KVStore` is a simple key-value store used to store and retrieve data. A `CommitKVStore` is a `KVStore` that also implements a `Committer`. By default, stores mounted in `baseapp`'s main `CommitMultiStore` are `CommitKVStore`s. The `KVStore` interface is primarily used to restrict modules from accessing the committer.

Individual `KVStore`s are used by modules to manage a subset of the global state. `KVStores` can be accessed by objects that hold a specific key. This `key` should only be exposed to the [`keeper`](#) of the module that defines the store.

`CommitKVStore`s are declared by proxy of their respective `key` and mounted on the application's `multistore` in the [main application file](#). In the same file, the `key` is also passed to the module's `keeper` that is responsible for managing the store.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/store.go#L229-L266
```

Apart from the traditional `Get` and `Set` methods, that a `KVStore` must implement via the `BasicKVStore` interface; a `KVStore` must provide an `Iterator(start, end)` method which returns an `Iterator` object. It is used to iterate over a range of keys, typically keys that share a common prefix. Below is an example from the bank's module keeper, used to iterate over all account balances:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/bank/keeper/view.go#L125-L140
```

IAVL Store

The default implementation of `KVStore` and `CommitKVStore` used in `baseapp` is the `iavl.Store`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/iavl/store.go#L35-L40
```

`iavl` stores are based around an [IAVL Tree](#), a self-balancing binary tree which guarantees that:

- `Get` and `Set` operations are $O(\log n)$, where n is the number of elements in the tree.
- Iteration efficiently returns the sorted elements within the range.
- Each tree version is immutable and can be retrieved even after a commit (depending on the pruning settings).

The documentation on the IAVL Tree is located [here](#).

DbAdapter Store

`dbadapter.Store` is a adapter for `dbm.DB` making it fulfilling the `KVStore` interface.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/dbadapter/store.go#L13-L16
```

`dbadapter.Store` embeds `dbm.DB`, meaning most of the `KVStore` interface functions are implemented. The other functions (mostly miscellaneous) are manually implemented. This store is primarily used within [Transient Stores](#)

Transient Store

`Transient.Store` is a base-layer `KVStore` which is automatically discarded at the end of the block.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/transient/store.go#L16-L19
```

`Transient.Store` is a `dbadapter.Store` with a `dbm.NewMemDB()`. All `KVStore` methods are reused. When `Store.Commit()` is called, a new `dbadapter.Store` is assigned, discarding previous reference and making it garbage collected.

This type of store is useful to persist information that is only relevant per-block. One example would be to store parameter changes (i.e. a bool set to `true` if a parameter changed in a block).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/params/types/subspace.go#L21-L31
```

Transient stores are typically accessed via the [context](#) via the `TransientStore()` method:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/context.go#L340-L343
```

KVStore Wrappers

CacheKVStore

`cachekv.Store` is a wrapper `KVStore` which provides buffered writing / cached reading functionalities over the underlying `KVStore`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/cachekv/store.go#L26-L36
```

This is the type used whenever an IAVL Store needs to be branched to create an isolated store (typically when we need to mutate a state that might be reverted later).

Get

`Store.Get()` firstly checks if `Store.cache` has an associated value with the key. If the value exists, the function returns it. If not, the function calls `Store.parent.Get()`, caches the result in `Store.cache`, and returns it.

Set

`Store.Set()` sets the key-value pair to the `Store.cache`. `cValue` has the field `dirty` bool which indicates whether the cached value is different from the underlying value. When `Store.Set()` caches a new pair, the `cValue.dirty` is set `true` so when `Store.Write()` is called it can be written to the underlying store.

Iterator

`Store.Iterator()` have to traverse on both cached items and the original items. In `Store.iterator()`, two iterators are generated for each of them, and merged. `memIterator` is essentially a slice of the `KVPairs`, used for cached items. `mergeIterator` is a combination of two iterators, where traverse happens ordered on both iterators.

GasKv Store

Cosmos SDK applications use `gas` to track resources usage and prevent spam. [`GasKv.Store`](#) is a `KVStore` wrapper that enables automatic gas consumption each time a read or write to the store is made. It is the solution of choice to track storage usage in Cosmos SDK applications.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/gaskv/store.go#L11-L17
```

When methods of the parent `KVStore` are called, `GasKv.Store` automatically consumes appropriate amount of gas depending on the `Store.gasConfig`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/gas.go#L219-L228
```

By default, all `KVStores` are wrapped in `GasKv.Stores` when retrieved. This is done in the `KVStore()` method of the [context](#):

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/context.go#L335-L338
```

In this case, the gas configuration set in the `context` is used. The gas configuration can be set using the `WithKVGasConfig` method of the `context`. Otherwise it uses the following default:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/types/gas.go#L230-L241
```

TraceKv Store

`tracekv.Store` is a wrapper `KVStore` which provides operation tracing functionalities over the underlying `KVStore`. It is applied automatically by the Cosmos SDK on all `KVStore` if tracing is enabled on the parent `MultiStore`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/tracekv/store.go#L20-L43
```

When each `KVStore` methods are called, `tracekv.Store` automatically logs `traceOperation` to the `Store.writer`. `traceOperation.Metadata` is filled with `Store.context` when it is not nil. `TraceContext` is a `map[string]interface{}`.

Prefix Store

`prefix.Store` is a wrapper `KVStore` which provides automatic key-prefixing functionalities over the underlying `KVStore`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/prefix/store.go#L15-L21
```

When `Store.{Get, Set}()` is called, the store forwards the call to its parent, with the key prefixed with the `Store.prefix`.

When `Store.Iterator()` is called, it does not simply prefix the `Store.prefix`, since it does not work as intended. In that case, some of the elements are traversed even they are not starting with the prefix.

ListenKv Store

`listenkv.Store` is a wrapper `KVStore` which provides state listening capabilities over the underlying `KVStore`. It is applied automatically by the Cosmos SDK on any `KVStore` whose `StoreKey` is specified

during state streaming configuration. Additional information about state streaming configuration can be found in the [store/streaming/README.md](#).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/store/listenkv/store.go#L11-L18
```

When `KVStore.Set` or `KVStore.Delete` methods are called, `listenkv.Store` automatically writes the operations to the set of `Store.listeners`.

BasicKVStore interface

An interface providing only the basic CRUD functionality (`Get` , `Set` , `Has` , and `Delete` methods), without iteration or caching. This is used to partially expose components of a larger store.

sidebar_position: 1

Encoding

:::note Synopsis While encoding in the Cosmos SDK used to be mainly handled by `go-amino` codec, the Cosmos SDK is moving towards using `gogoproobuf` for both state and client-side encoding. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK application](#)

:::

Encoding

The Cosmos SDK utilizes two binary wire encoding protocols, [Amino](#) which is an object encoding specification and [Protocol Buffers](#), a subset of Proto3 with an extension for interface support. See the [Proto3 spec](#) for more information on Proto3, which Amino is largely compatible with (but not with Proto2).

Due to Amino having significant performance drawbacks, being reflection-based, and not having any meaningful cross-language/client support, Protocol Buffers, specifically [gogoproobuf](#), is being used in place of Amino. Note, this process of using Protocol Buffers over Amino is still an ongoing process.

Binary wire encoding of types in the Cosmos SDK can be broken down into two main categories, client encoding and store encoding. Client encoding mainly revolves around transaction processing and signing, whereas store encoding revolves around types used in state-machine transitions and what is ultimately stored in the Merkle tree.

For store encoding, protobuf definitions can exist for any type and will typically have an Amino-based "intermediary" type. Specifically, the protobuf-based type definition is used for serialization and persistence, whereas the Amino-based type is used for business logic in the state-machine where they may convert back-and-forth. Note, the Amino-based types may slowly be phased-out in the future, so developers should take note to use the protobuf message definitions where possible.

In the `codec` package, there exists two core interfaces, `BinaryCodec` and `JSONCodec`, where the former encapsulates the current Amino interface except it operates on types implementing the latter instead of generic `interface{}` types.

In addition, there exists two implementations of `Codec`. The first being `AminoCodec`, where both binary and JSON serialization is handled via Amino. The second being `ProtoCodec`, where both binary and JSON serialization is handled via Protobuf.

This means that modules may use Amino or Protobuf encoding, but the types must implement `ProtoMarshaler`. If modules wish to avoid implementing this interface for their types, they may use an Amino codec directly.

Amino

Every module uses an Amino codec to serialize types and interfaces. This codec typically has types and interfaces registered in that module's domain only (e.g. messages), but there are exceptions like `x/gov`. Each module exposes a `RegisterLegacyAminoCodec` function that allows a user to provide a codec and have all the types registered. An application will call this method for each necessary module.

Where there is no protobuf-based type definition for a module (see below), Amino is used to encode and decode raw wire bytes to the concrete type or interface:

```
bz := keeper.cdc.MustMarshal(typeOrInterface)
keeper.cdc.MustUnmarshal(bz, &typeOrInterface)
```

Note, there are length-prefixed variants of the above functionality and this is typically used for when the data needs to be streamed or grouped together (e.g. `ResponseDeliverTx.Data`)

Authz authorizations and Gov/Group proposals

Since authz's `MsgExec` and `MsgGrant` message types, as well as gov's and group's `MsgSubmitProposal`, can contain different messages instances, it is important that developers add the following code inside the `init` method of their module's `codec.go` file:

```
import (
    authzcodec "github.com/cosmos/cosmos-sdk/x/authz/codec"
    govcodec "github.com/cosmos/cosmos-sdk/x/gov/codec"
    groupcodec "github.com/cosmos/cosmos-sdk/x/group/codec"
)

init() {
    // Register all Amino interfaces and concrete types on the authz and gov Amino
    // codec so that this can later be
    // used to properly serialize MsgGrant, MsgExec and MsgSubmitProposal instances
    RegisterLegacyAminoCodec(authzcodec.Amino)
    RegisterLegacyAminoCodec(govcodec.Amino)
    RegisterLegacyAminoCodec(groupcodec.Amino)
}
```

This will allow the `x/authz` module to properly serialize and de-serializes `MsgExec` instances using Amino, which is required when signing this kind of messages using a Ledger.

Gogoproto

Modules are encouraged to utilize Protobuf encoding for their respective types. In the Cosmos SDK, we use the [Gogoproto](#) specific implementation of the Protobuf spec that offers speed and DX improvements compared to the official [Google protobuf implementation](#).

Guidelines for protobuf message definitions

In addition to [following official Protocol Buffer guidelines](#), we recommend using these annotations in .proto files when dealing with interfaces:

- use `cosmos_proto.accepts_interface` to annotate `Any` fields that accept interfaces
 - pass the same fully qualified name as `protoName` to`InterfaceRegistry.RegisterInterface`
 - example: `(cosmos_proto.accepts_interface) = "cosmos.gov.v1beta1.Content"`
(and not just `Content`)
- annotate interface implementations with `cosmos_proto.implements_interface`
 - pass the same fully qualified name as `protoName` to`InterfaceRegistry.RegisterInterface`
 - example: `(cosmos_proto.implements_interface) = "cosmos.authz.v1beta1.Authorization"` (and not just `Authorization`)

Code generators can then match the `accepts_interface` and `implements_interface` annotations to know whether some Protobuf messages are allowed to be packed in a given `Any` field or not.

Transaction Encoding

Another important use of Protobuf is the encoding and decoding of [transactions](#). Transactions are defined by the application or the Cosmos SDK but are then passed to the underlying consensus engine to be relayed to other peers. Since the underlying consensus engine is agnostic to the application, the consensus engine accepts only transactions in the form of raw bytes.

- The `TxEncoder` object performs the encoding.
- The `TxDecoder` object performs the decoding.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/tx\_msg.go#L91-L95
```

A standard implementation of both these objects can be found in the [auth/tx module](#):

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/tx/decoder.go
```

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/auth/tx/encoder.go
```

See [ADR-020](#) for details of how a transaction is encoded.

Interface Encoding and Usage of `Any`

The Protobuf DSL is strongly typed, which can make inserting variable-typed fields difficult. Imagine we want to create a `Profile` protobuf message that serves as a wrapper over [an account](#):

```

message Profile {
    // account is the account associated to a profile.
    cosmos.auth.v1beta1.BaseAccount account = 1;
    // bio is a short description of the account.
    string bio = 4;
}

```

In this `Profile` example, we hardcoded `account` as a `BaseAccount`. However, there are several other types of [user accounts related to vesting](#), such as `BaseVestingAccount` or `ContinuousVestingAccount`. All of these accounts are different, but they all implement the `AccountI` interface. How would you create a `Profile` that allows all these types of accounts with an `account` field that accepts an `AccountI` interface?

<https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/account.go#L15-L32>

In [ADR-019](#), it has been decided to use `Any`s to encode interfaces in protobuf. An `Any` contains an arbitrary serialized message as bytes, along with a URL that acts as a globally unique identifier for and resolves to that message's type. This strategy allows us to pack arbitrary Go types inside protobuf messages. Our new `Profile` then looks like:

```

message Profile {
    // account is the account associated to a profile.
    google.protobuf.Any account = 1 [
        (cosmos_proto.accepts_interface) = "cosmos.auth.v1beta1.AccountI"; // Asserts
        that this field only accepts Go types implementing `AccountI`. It is purely
        informational for now.
    ];
    // bio is a short description of the account.
    string bio = 4;
}

```

To add an account inside a profile, we need to "pack" it inside an `Any` first, using `codectypes.NewAnyWithValue`:

```

var myAccount AccountI
myAccount = ... // Can be a BaseAccount, a ContinuousVestingAccount or any struct
implementing `AccountI`


// Pack the account into an Any
accAny, err := codectypes.NewAnyWithValue(myAccount)
if err != nil {
    return nil, err
}

// Create a new Profile with the any.
profile := Profile {
    Account: accAny,
    Bio: "some bio",
}

```

```

}

// We can then marshal the profile as usual.
bz, err := cdc.Marshal(profile)
jsonBz, err := cdc.MarshalJSON(profile)

```

To summarize, to encode an interface, you must 1/ pack the interface into an `Any` and 2/ marshal the `Any`. For convenience, the Cosmos SDK provides a `MarshalInterface` method to bundle these two steps. Have a look at [a real-life example in the `x/auth` module](#).

The reverse operation of retrieving the concrete Go type from inside an `Any`, called "unpacking", is done with the `GetCachedValue()` on `Any`.

```

profileBz := ... // The proto-encoded bytes of a Profile, e.g. retrieved through
gRPC.

var myProfile Profile
// Unmarshal the bytes into the myProfile struct.
err := cdc.Unmarshal(profileBz, &myProfile)

// Let's see the types of the Account field.
fmt.Printf("%T\n", myProfile.Account) // Prints "Any"
fmt.Printf("%T\n", myProfile.Account.GetCachedValue()) // Prints "BaseAccount",
"ContinuousVestingAccount" or whatever was initially packed in the Any.

// Get the address of the accountt.
accAddr := myProfile.Account.GetCachedValue().(AccountI).GetAddress()

```

It is important to note that for `GetCachedValue()` to work, `Profile` (and any other structs embedding `Profile`) must implement the `UnpackInterfaces` method:

```

func (p *Profile) UnpackInterfaces(unpacker codectypes.AnyUnpacker) error {
    if p.Account != nil {
        var account AccountI
        return unpacker.UnpackAny(p.Account, &account)
    }

    return nil
}

```

The `UnpackInterfaces` gets called recursively on all structs implementing this method, to allow all `Any`'s to have their `GetCachedValue()` correctly populated.

For more information about interface encoding, and especially on `UnpackInterfaces` and how the `Any`'s `type_url` gets resolved using the `InterfaceRegistry`, please refer to [ADR-019](#).

`Any` Encoding in the Cosmos SDK

The above `Profile` example is a fictive example used for educational purposes. In the Cosmos SDK, we use `Any` encoding in several places (non-exhaustive list):

- the `cryptotypes.PubKey` interface for encoding different types of public keys,

- the `sdk.Msg` interface for encoding different `Msg`'s in a transaction,
- the `AccountI` interface for encoding different types of accounts (similar to the above example) in the `x/auth` query responses,
- the `EvidenceI` interface for encoding different types of evidences in the `x/evidence` module,
- the `AuthorizationI` interface for encoding different types of `x/authz` authorizations,
- the [Validator](#) struct that contains information about a validator.

A real-life example of encoding the pubkey as `Any` inside the `Validator` struct in `x/staking` is shown in the following example:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/x/staking/types/validator.go#L41-L64
```

Any's TypeURL

When packing a protobuf message inside an `Any`, the message's type is uniquely defined by its type URL, which is the message's fully qualified name prefixed by a `/` (slash) character. In some implementations of `Any`, like the gogoproto one, there's generally [a resolvable prefix, e.g. `type.googleapis.com`](#). However, in the Cosmos SDK, we made the decision to not include such prefix, to have shorter type URLs. The Cosmos SDK's own `Any` implementation can be found in `github.com/cosmos/cosmos-
sdk/codec/types`.

The Cosmos SDK is also switching away from gogoproto to the official `google.golang.org/protobuf` (known as the Protobuf API v2). Its default `Any` implementation also contains the [`type.googleapis.com`](#) prefix. To maintain compatibility with the SDK, the following methods from `"google.golang.org/protobuf/types/known/anypb"` should not be used:

- `anypb.New`
- `anypb.MarshalFrom`
- `anypb.Any#MarshalFrom`

Instead, the Cosmos SDK provides helper functions in `"github.com/cosmos/cosmos-proto/anyutil"`, which create an official `anypb.Any` without inserting the prefixes:

- `anyutil.New`
- `anyutil.MarshalFrom`

For example, to pack a `sdk.Msg` called `internalMsg`, use:

```
import (
-    "google.golang.org/protobuf/types/known/anypb"
+    "github.com/cosmos/cosmos-proto/anyutil"
)

- anyMsg, err := anypb.New(internalMsg.Message().Interface())
+ anyMsg, err := anyutil.New(internalMsg.Message().Interface())

- fmt.Println(anyMsg.TypeURL) // type.googleapis.com/cosmos.bank.v1beta1.MsgSend
+ fmt.Println(anyMsg.TypeURL) // /cosmos.bank.v1beta1.MsgSend
```

FAQ

How to create modules using protobuf encoding

Defining module types

Protobuf types can be defined to encode:

- state
- [Msg.S](#)
- [Query services](#)
- [genesis](#)

Naming and conventions

We encourage developers to follow industry guidelines: [Protocol Buffers style guide](#) and [Buf](#), see more details in [ADR 023](#)

How to update modules to protobuf encoding

If modules do not contain any interfaces (e.g. `Account` or `Content`), then they may simply migrate any existing types that are encoded and persisted via their concrete Amino codec to Protobuf (see 1. for further guidelines) and accept a `Marshaler` as the codec which is implemented via the `ProtoCodec` without any further customization.

However, if a module type composes an interface, it must wrap it in the `sdk.Any` (from `/types` package) type. To do that, a module-level .proto file must use [google.protobuf.Any](#) for respective message type interface types.

For example, in the `x/evidence` module defines an `Evidence` interface, which is used by the `MsgSubmitEvidence`. The structure definition must use `sdk.Any` to wrap the evidence file. In the proto file we define it as follows:

```
// proto/cosmos/evidence/v1beta1/tx.proto

message MsgSubmitEvidence {
    string submitter = 1;
    google.protobuf.Any evidence = 2 [(cosmos_proto.accepts_interface) =
"cosmos.evidence.v1beta1.Evidence"];
}
```

The Cosmos SDK `codec.Codec` interface provides support methods `MarshalInterface` and `UnmarshalInterface` to easy encoding of state to `Any`.

Module should register interfaces using `InterfaceRegistry` which provides a mechanism for registering interfaces: `RegisterInterface(protoName string, iface interface{}, impls ...proto.Message)` and implementations: `RegisterImplementations(iface interface{}, impls ...proto.Message)` that can be safely unpacked from Any, similarly to type registration with Amino:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/codec/types/interface\_registry.go#L28-L75
```

In addition, an `UnpackInterfaces` phase should be introduced to deserialization to unpack interfaces before they're needed. Protobuf types that contain a protobuf `Any` either directly or via one of their members should implement the `UnpackInterfacesMessage` interface:

```
type UnpackInterfacesMessage interface {
    UnpackInterfaces(InterfaceUnpacker) error
}
```

sidebar_position: 1

gRPC, REST, and CometBFT Endpoints

:::note Synopsis This document presents an overview of all the endpoints a node exposes: gRPC, REST as well as some other endpoints. :::

An Overview of All Endpoints

Each node exposes the following endpoints for users to interact with a node, each endpoint is served on a different port. Details on how to configure each endpoint is provided in the endpoint's own section.

- the gRPC server (default port: `9090`),
- the REST server (default port: `1317`),
- the CometBFT RPC endpoint (default port: `26657`).

:::tip The node also exposes some other endpoints, such as the CometBFT P2P endpoint, or the [Prometheus endpoint](#), which are not directly related to the Cosmos SDK. Please refer to the [CometBFT documentation](#) for more information about these endpoints. :::

gRPC Server

In the Cosmos SDK, Protobuf is the main [encoding](#) library. This brings a wide range of Protobuf-based tools that can be plugged into the Cosmos SDK. One such tool is [gRPC](#), a modern open-source high performance RPC framework that has decent client support in several languages.

Each module exposes a [Protobuf Query service](#) that defines state queries. The `Query` services and a transaction service used to broadcast transactions are hooked up to the gRPC server via the following function inside the application:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/server/types/app.go#L46-L48
```

Note: It is not possible to expose any [Protobuf Msg service](#) endpoints via gRPC. Transactions must be generated and signed using the CLI or programmatically before they can be broadcasted using gRPC. See [Generating, Signing, and Broadcasting Transactions](#) for more information.

The `grpc.Server` is a concrete gRPC server, which spawns and serves all gRPC query requests and a broadcast transaction request. This server can be configured inside `~/.simapp/config/app.toml`:

- `grpc.enable = true|false` field defines if the gRPC server should be enabled. Defaults to `true`.
- `grpc.address = {string}` field defines the `ip:port` the server should bind to. Defaults to `localhost:9090`.

:::tip `~/.simapp` is the directory where the node's configuration and databases are stored. By default, it's set to `~/.{app_name}` .:::

Once the gRPC server is started, you can send requests to it using a gRPC client. Some examples are given in our [Interact with the Node](#) tutorial.

An overview of all available gRPC endpoints shipped with the Cosmos SDK is [Protobuf documentation](#).

REST Server

Cosmos SDK supports REST routes via gRPC-gateway.

All routes are configured under the following fields in `~/.simapp/config/app.toml` :

- `api.enable = true|false` field defines if the REST server should be enabled. Defaults to `false`.
- `api.address = {string}` field defines the `ip:port` the server should bind to. Defaults to `tcp://localhost:1317`.
- some additional API configuration options are defined in `~/.simapp/config/app.toml` , along with comments, please refer to that file directly.

gRPC-gateway REST Routes

If, for various reasons, you cannot use gRPC (for example, you are building a web application, and browsers don't support HTTP2 on which gRPC is built), then the Cosmos SDK offers REST routes via gRPC-gateway.

[gRPC-gateway](#) is a tool to expose gRPC endpoints as REST endpoints. For each gRPC endpoint defined in a Protobuf `Query` service, the Cosmos SDK offers a REST equivalent. For instance, querying a balance could be done via the `/cosmos.bank.v1beta1.QueryAllBalances` gRPC endpoint, or alternatively via the gRPC-gateway `"/cosmos/bank/v1beta1/balances/{address}"` REST endpoint: both will return the same result. For each RPC method defined in a Protobuf `Query` service, the corresponding REST endpoint is defined as an option:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-
alpha.0/proto/cosmos/bank/v1beta1/query.proto#L23-L30
```

For application developers, gRPC-gateway REST routes needs to be wired up to the REST server, this is done by calling the `RegisterGRPCGatewayRoutes` function on the `ModuleManager`.

Swagger

A [Swagger](#) (or OpenAPIv2) specification file is exposed under the `/swagger` route on the API server.

Swagger is an open specification describing the API endpoints a server serves, including description, input arguments, return types and much more about each endpoint.

Enabling the `/swagger` endpoint is configurable inside `~/.simapp/config/app.toml` via the `api.swagger` field, which is set to true by default.

For application developers, you may want to generate your own Swagger definitions based on your custom modules. The Cosmos SDK's [Swagger generation script](#) is a good place to start.

CometBFT RPC

Independently from the Cosmos SDK, CometBFT also exposes a RPC server. This RPC server can be configured by tuning parameters under the `rpc` table in the `~/.simapp/config/config.toml`, the default listening address is `tcp://localhost:26657`. An OpenAPI specification of all CometBFT RPC endpoints is available [here](#).

Some CometBFT RPC endpoints are directly related to the Cosmos SDK:

- `/abci_query` : this endpoint will query the application for state. As the `path` parameter, you can send the following strings:
 - any Protobuf fully-qualified service method, such as `/cosmos.bank.v1beta1.Query/AllBalances`. The `data` field should then include the method's request parameter(s) encoded as bytes using Protobuf.
 - `/app/simulate` : this will simulate a transaction, and return some information such as gas used.
 - `/app/version` : this will return the application's version.
 - `/store/{path}` : this will query the store directly.
 - `/p2p/filter/addr/{port}` : this will return a filtered list of the node's P2P peers by address port.
 - `/p2p/filter/id/{id}` : this will return a filtered list of the node's P2P peers by ID.
- `/broadcast_tx_{async,async,commit}` : these 3 endpoint will broadcast a transaction to other peers. CLI, gRPC and REST expose [a way to broadcast transactions](#), but they all use these 3 CometBFT RPCs under the hood.

Comparison Table

Name	Advantages	Disadvantages
gRPC	<ul style="list-style-type: none">- can use code-generated stubs in various languages- supports streaming and bidirectional communication (HTTP2)- small wire binary sizes, faster transmission	<ul style="list-style-type: none">- based on HTTP2, not available in browsers- learning curve (mostly due to Protobuf)
REST	<ul style="list-style-type: none">- ubiquitous- client libraries in all languages, faster implementation	<ul style="list-style-type: none">- only supports unary request-response communication (HTTP1.1)- bigger over-the-wire message sizes (JSON)
CometBFT RPC	<ul style="list-style-type: none">- easy to use	<ul style="list-style-type: none">- bigger over-the-wire message sizes (JSON)

sidebar_position: 1

Command-Line Interface

:::note Synopsis This document describes how command-line interface (CLI) works on a high-level, for an [application](#). A separate document for implementing a CLI for a Cosmos SDK [module](#) can be found [here](#). :::

Command-Line Interface

Example Command

There is no set way to create a CLI, but Cosmos SDK modules typically use the [Cobra Library](#). Building a CLI with Cobra entails defining commands, arguments, and flags. [Commands](#) understand the actions users wish to take, such as `tx` for creating a transaction and `query` for querying the application. Each command can also have nested subcommands, necessary for naming the specific transaction type. Users also supply [Arguments](#), such as account numbers to send coins to, and [Flags](#) to modify various aspects of the commands, such as gas prices or which node to broadcast to.

Here is an example of a command a user might enter to interact with the simapp CLI `simd` in order to send some tokens:

```
simd tx bank send $MY_VALIDATOR_ADDRESS $RECIPIENT 1000stake --gas auto --gas-prices
<gasPrices>
```

The first four strings specify the command:

- The root command for the entire application `simd`.
- The subcommand `tx`, which contains all commands that let users create transactions.
- The subcommand `bank` to indicate which module to route the command to ([x/bank](#) module in this case).
- The type of transaction `send`.

The next two strings are arguments: the `from_address` the user wishes to send from, the `to_address` of the recipient, and the `amount` they want to send. Finally, the last few strings of the command are optional flags to indicate how much the user is willing to pay in fees (calculated using the amount of gas used to execute the transaction and the gas prices provided by the user).

The CLI interacts with a [node](#) to handle this command. The interface itself is defined in a `main.go` file.

Building the CLI

The `main.go` file needs to have a `main()` function that creates a root command, to which all the application commands will be added as subcommands. The root command additionally handles:

- **setting configurations** by reading in configuration files (e.g. the Cosmos SDK config file).
- **adding any flags** to it, such as `--chain-id`.
- **instantiating the codec** by injecting the application codecs. The `codec` is used to encode and decode data structures for the application - stores can only persist `[]byte`s so the developer must define a serialization format for their data structures or use the default, Protobuf.
- **adding subcommand** for all the possible user interactions, including [transaction commands](#) and [query commands](#).

The `main()` function finally creates an executor and [execute](#) the root command. See an example of `main()` function from the `simapp` application:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/main.go#L12-L24
```

The rest of the document will detail what needs to be implemented for each step and include smaller portions of code from the `simapp` CLI files.

Adding Commands to the CLI

Every application CLI first constructs a root command, then adds functionality by aggregating subcommands (often with further nested subcommands) using `rootCmd.AddCommand()`. The bulk of an application's unique capabilities lies in its transaction and query commands, called `TxCmd` and `QueryCmd` respectively.

Root Command

The root command (called `rootCmd`) is what the user first types into the command line to indicate which application they wish to interact with. The string used to invoke the command (the "Use" field) is typically the name of the application suffixed with `-d`, e.g. `simd` or `gaiad`. The root command typically includes the following commands to support basic functionality in the application.

- **Status** command from the Cosmos SDK rpc client tools, which prints information about the status of the connected [Node](#). The Status of a node includes `NodeInfo`, `SyncInfo` and `ValidatorInfo`.
- **Keys** [commands](#) from the Cosmos SDK client tools, which includes a collection of subcommands for using the key functions in the Cosmos SDK crypto tools, including adding a new key and saving it to the keyring, listing all public keys stored in the keyring, and deleting a key. For example, users can type `simd keys add <name>` to add a new key and save an encrypted copy to the keyring, using the flag `--recover` to recover a private key from a seed phrase or the flag `--multisig` to group multiple keys together to create a multisig key. For full details on the `add` key command, see the code [here](#). For more details about usage of `--keyring-backend` for storage of key credentials look at the [keyring docs](#).
- **Server** commands from the Cosmos SDK server package. These commands are responsible for providing the mechanisms necessary to start an ABCI CometBFT application and provides the CLI framework (based on [cobra](#)) necessary to fully bootstrap an application. The package exposes two core functions: `StartCmd` and `ExportCmd` which creates commands to start the application and export state respectively. Learn more [here](#).
- [Transaction](#) commands.
- [Query](#) commands.

Next is an example `rootCmd` function from the `simapp` application. It instantiates the root command, adds a [persistent flag](#) and `PreRun` function to be run before every execution, and adds all of the necessary subcommands.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L47-L130
```

:::tip Use the `EnhanceRootCommand()` from the AutoCLI options to automatically add auto-generated commands from the modules to the root command. Additionnally it adds all manually defined modules commands (`tx` and `query`) as well. Read more about [AutoCLI](#) in its dedicated section. :::

`rootCmd` has a function called `initAppConfig()` which is useful for setting the application's custom configs. By default app uses CometBFT app config template from Cosmos SDK, which can be over-written via `initAppConfig()`. Here's an example code to override default `app.toml` template.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L144-L199
```

The `initAppConfig()` also allows overriding the default Cosmos SDK's [server config](#). One example is the `min-gas-prices` config, which defines the minimum gas prices a validator is willing to accept for processing a transaction. By default, the Cosmos SDK sets this parameter to `""` (empty string), which forces all validators to tweak their own `app.toml` and set a non-empty value, or else the node will halt on startup. This might not be the best UX for validators, so the chain developer can set a default `app.toml` value for validators inside this `initAppConfig()` function.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L164-L180
```

The root-level `status` and `keys` subcommands are common across most applications and do not interact with application state. The bulk of an application's functionality - what users can actually *do* with it - is enabled by its `tx` and `query` commands.

Transaction Commands

[Transactions](#) are objects wrapping [Msg](#)s that trigger state changes. To enable the creation of transactions using the CLI interface, a function `txCommand` is generally added to the `rootCmd`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L222-L229
```

This `txCommand` function adds all the transaction available to end-users for the application. This typically includes:

- **Sign command** from the `auth` module that signs messages in a transaction. To enable multisig, add the `auth` module's `MultiSign` command. Since every transaction requires some sort of signature in order to be valid, the signing command is necessary for every application.
- **Broadcast command** from the Cosmos SDK client tools, to broadcast transactions.
- All [module transaction commands](#) the application is dependent on, retrieved by using the [basic module manager's](#) `AddTxCommands()` function, or enhanced by [AutoCLI](#).

Here is an example of a `txCommand` aggregating these subcommands from the `simapp` application:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L270-L292
```

:::tip When using AutoCLI to generate module transaction commands, `EnhanceRootCommand()` automatically adds the module `tx` command to the root command. Read more about [AutoCLI](#) in its dedicated section. :::

Query Commands

[Queries](#) are objects that allow users to retrieve information about the application's state. To enable the creation of queries using the CLI interface, a function `queryCommand` is generally added to the `rootCmd`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L222-L229
```

This `queryCommand` function adds all the queries available to end-users for the application. This typically includes:

- **QueryTx** and/or other transaction query commands from the `auth` module which allow the user to search for a transaction by inputting its hash, a list of tags, or a block height. These queries allow users to see if transactions have been included in a block.
- **Account command** from the `auth` module, which displays the state (e.g. account balance) of an account given an address.
- **Validator command** from the Cosmos SDK rpc client tools, which displays the validator set of a given height.
- **Block command** from the Cosmos SDK RPC client tools, which displays the block data for a given height.
- All [module query commands](#) the application is dependent on, retrieved by using the [basic module manager's](#) `AddQueryCommands()` function, or enhanced by [AutoCLI](#).

Here is an example of a `queryCommand` aggregating subcommands from the `simapp` application:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root_v2.go#L249-L268
```

:::tip When using AutoCLI to generate module query commands, `EnhanceRootCommand()` automatically adds the module `query` command to the root command. Read more about [AutoCLI](#) in its dedicated section. :::

Flags

Flags are used to modify commands; developers can include them in a `flags.go` file with their CLI. Users can explicitly include them in commands or pre-configure them by inside their [app.toml](#). Commonly pre-configured flags include the `--node` to connect to and `--chain-id` of the blockchain the user wishes to interact with.

A *persistent* flag (as opposed to a *local* flag) added to a command transcends all of its children: subcommands will inherit the configured values for these flags. Additionally, all flags have default values when they are added to commands; some toggle an option off but others are empty values that the user needs to override to create valid commands. A flag can be explicitly marked as *required* so that an error is automatically thrown if the user does not provide a value, but it is also acceptable to handle unexpected missing flags differently.

Flags are added to commands directly (generally in the [module's CLI file](#) where module commands are defined) and no flag except for the `rootCmd` persistent flags has to be added at application level. It is common to add a *persistent* flag for `--chain-id`, the unique identifier of the blockchain the application pertains to, to the root command. Adding this flag can be done in the `main()` function. Adding this flag makes sense as the chain ID should not be changing across commands in this application CLI.

Environment variables

Each flag is bound to its respective named environment variable. Then name of the environment variable consists of two parts - capital case `basename` followed by flag name of the flag. `-` must be substituted with `_`. For example flag `--home` for application with basename `GAIA` is bound to `GAIA_HOME`. It allows reducing the amount of flags typed for routine operations. For example instead of:

```
gaia --home=./ --node=<node address> --chain-id="testchain-1" --keyring-backend=test
tx ... --from=<key name>
```

this will be more convenient:

```
# define env variables in .env, .envrc etc
GAIA_HOME=<path to home>
GAIA_NODE=<node address>
GAIA_CHAIN_ID="testchain-1"
GAIA_KEYRING_BACKEND="test"

# and later just use
gaia tx ... --from=<key name>
```

Configurations

It is vital that the root command of an application uses `PersistentPreRun()` cobra command property for executing the command, so all child commands have access to the server and client contexts. These contexts are set as their default values initially and maybe modified, scoped to the command, in their respective `PersistentPreRun()` functions. Note that the `client.Context` is typically pre-populated with "default" values that may be useful for all commands to inherit and override if necessary.

Here is an example of an `PersistentPreRun()` function from `simapp`:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/simd/cmd/root\_v2.go#L81-L120
```

The `SetCmdClientContextHandler` call reads persistent flags via `ReadPersistentCommandFlags` which creates a `client.Context` and sets that on the root command's `Context`.

The `InterceptConfigsPreRunHandler` call creates a viper literal, default `server.Context`, and a logger and sets that on the root command's `Context`. The `server.Context` will be modified and saved to disk. The internal `interceptConfigs` call reads or creates a CometBFT configuration based on the home path provided. In addition, `interceptConfigs` also reads and loads the application configuration,

`app.toml`, and binds that to the `server.Context` viper literal. This is vital so the application can get access to not only the CLI flags, but also to the application configuration values provided by this file.

:::tip When willing to configure which logger is used, do not to use `InterceptConfigsPreRunHandler`, which sets the default SDK logger, but instead use `InterceptConfigsAndCreateContext` and set the server context and the logger manually:

```
-return server.InterceptConfigsPreRunHandler(cmd, customAppTemplate,
customAppConfig, customCMTConfig)

+serverCtx, err := server.InterceptConfigsAndCreateContext(cmd, customAppTemplate,
customAppConfig, customCMTConfig)
+if err != nil {
+    return err
+}

+// overwrite default server logger
+logger, err := server.CreateSDKLogger(serverCtx, cmd.OutOrStdout())
+if err != nil {
+    return err
+}
+serverCtx.Logger = logger.With(log.ModuleKey, "server")

+// set server context
+return server.SetCmdServerContext(cmd, serverCtx)
```

:::

sidebar_position: 1

Events

:::note Synopsis `Event`s are objects that contain information about the execution of the application. They are mainly used by service providers like block explorers and wallet to track the execution of various messages and index transactions. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK application](#)
- [CometBFT Documentation on Events](#)

:::

Events

Events are implemented in the Cosmos SDK as an alias of the ABCI `Event` type and take the form of:

```
{eventType}.{attributeKey}={attributeValue} .
```

```
https://github.com/cometbft/cometbft/blob/v0.37.0/proto/tendermint/abci/types.proto#L3  
L343
```

An Event contains:

- A `type` to categorize the Event at a high-level; for example, the Cosmos SDK uses the `"message"` type to filter Events by `Msg`s.
- A list of `attributes` are key-value pairs that give more information about the categorized Event. For example, for the `"message"` type, we can filter Events by key-value pairs using `message.action={some_action}`, `message.module={some_module}` or `message.sender={some_sender}`.
- A `msg_index` to identify which messages relate to the same transaction

:::tip To parse the attribute values as strings, make sure to add `'` (single quotes) around each attribute value. :::

Typed Events are Protobuf-defined [messages](#) used by the Cosmos SDK for emitting and querying Events. They are defined in a `event.proto` file, on a **per-module basis** and are read as `proto.Message`. *Legacy Events* are defined on a **per-module basis** in the module's `/types/events.go` file. They are triggered from the module's Protobuf [Msg service](#) by using the [EventManager](#).

In addition, each module documents its events under in the `Events` sections of its specs (`x/{moduleName}/ README.md`).

Lastly, Events are returned to the underlying consensus engine in the response of the following ABCI messages:

- [BeginBlock](#)
- [EndBlock](#)
- [CheckTx](#)
- [Transaction Execution](#)

Examples

The following examples show how to query Events using the Cosmos SDK.

Event	Description
<code>tx.height=23</code>	Query all transactions at height 23
<code>message.action='/cosmos.bank.v1beta1.Msg/Send'</code>	Query all transactions containing a x/bank Send Service Msg . Note the <code>'s</code> around the value.
<code>message.module='bank'</code>	Query all transactions containing messages from the x/bank module. Note the <code>'s</code> around the value.
<code>create_validator.validator='cosmosval1...'</code>	x/staking-specific Event, see x/staking SPEC .

EventManager

In Cosmos SDK applications, Events are managed by an abstraction called the `EventManager`. Internally, the `EventManager` tracks a list of Events for the entire execution flow of `FinalizeBlock` (i.e.

transaction execution, `BeginBlock`, `EndBlock`).

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/events.go#L19-L26
```

The `EventManager` comes with a set of useful methods to manage Events. The method that is used most by module and application developers is `EmitTypedEvent` or `EmitEvent` that tracks an Event in the `EventManager`.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/types/events.go#L53-L62
```

Module developers should handle Event emission via the `EventManager#EmitTypedEvent` or `EventManager#EmitEvent` in each message `Handler` and in each `BeginBlock` / `EndBlock` handler. The `EventManager` is accessed via the [Context](#), where Event should be already registered, and emitted like this:

Typed events:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/x/group/keeper/msg\_server.go#L95-L97
```

Legacy events:

```
ctx.EventManager().EmitEvent(  
    sdk.NewEvent(eventType, sdk.NewAttribute(attributeKey, attributeValue)),  
)
```

Module's `handler` function should also set a new `EventManager` to the `context` to isolate emitted Events per message :

```
func NewHandler(keeper Keeper) sdk.Handler {  
    return func(ctx sdk.Context, msg sdk.Msg) (*sdk.Result, error) {  
        ctx = ctx.WithEventManager(sdk.NewEventManager())  
        switch msg := msg.(type) {
```

See the [Msg services](#) concept doc for a more detailed view on how to typically implement Events and use the `EventManager` in modules.

Subscribing to Events

You can use CometBFT's [Websocket](#) to subscribe to Events by calling the `subscribe` RPC method:

```
{  
    "jsonrpc": "2.0",  
    "method": "subscribe",  
    "id": "0",  
    "params": {  
        "query": "tm.event='eventCategory' AND  
        eventType.eventAttribute='attributeValue'"
```

```
    }
}
```

The main `eventCategory` you can subscribe to are:

- `NewBlock` : Contains Events triggered during `BeginBlock` and `EndBlock`.
- `Tx` : Contains Events triggered during `DeliverTx` (i.e. transaction processing).
- `ValidatorSetUpdates` : Contains validator set updates for the block.

These Events are triggered from the `state` package after a block is committed. You can get the full list of Event categories [on the CometBFT Go documentation](#).

The `type` and `attribute` value of the `query` allow you to filter the specific Event you are looking for.

For example, a `Mint` transaction triggers an Event of type `EventMint` and has an `Id` and an `Owner` as attributes (as defined in the [events.proto file of the NFT module](#)).

Subscribing to this Event would be done like so:

```
{
  "jsonrpc": "2.0",
  "method": "subscribe",
  "id": "0",
  "params": {
    "query": "tm.event='Tx' AND mint.owner='ownerAddress'"
  }
}
```

where `ownerAddress` is an address following the [AccAddress](#) format.

The same way can be used to subscribe to [legacy events](#).

Default Events

There are a few events that are automatically emitted for all messages, directly from `baseapp`.

- `message.action` : The name of the message type.
- `message.sender` : The address of the message signer.
- `message.module` : The name of the module that emitted the message.

:::tip The module name is assumed by `baseapp` to be the second element of the message route: `"cosmos.bank.v1beta1.MsgSend" -> "bank"`. In case a module does not follow the standard message path, (e.g. IBC), it is advised to keep emitting the module name event.

Baseapp only emits that event if the module have not already done so.

:::

sidebar_position: 1

Telemetry

:::note Synopsis Gather relevant insights about your application and modules with custom metrics and telemetry. :::

The Cosmos SDK enables operators and developers to gain insight into the performance and behavior of their application through the use of the `telemetry` package. To enable telemetrics, set

```
telemetry.enabled = true
```

in the `app.toml` config file.

The Cosmos SDK currently supports enabling in-memory and prometheus as telemetry sinks. In-memory sink is always attached (when the telemetry is enabled) with 10 second interval and 1 minute retention. This means that metrics will be aggregated over 10 seconds, and metrics will be kept alive for 1 minute.

To query active metrics (see retention note above) you have to enable API server (`api.enabled = true` in the `app.toml`). Single API endpoint is exposed: `http://localhost:1317/metrics?format={text|prometheus}`, the default being `text`.

Emitting metrics

If telemetry is enabled via configuration, a single global metrics collector is registered via the [go-metrics](#) library. This allows emitting and collecting metrics through simple [API](#). Example:

```
func EndBlocker(ctx sdk.Context, k keeper.Keeper) {
    defer telemetry.ModuleMeasureSince(types.ModuleName, time.Now(),
        telemetry.MetricKeyEndBlocker)

    // ...
}
```

Developers may use the `telemetry` package directly, which provides wrappers around metric APIs that include adding useful labels, or they must use the `go-metrics` library directly. It is preferable to add as much context and adequate dimensionality to metrics as possible, so the `telemetry` package is advised. Regardless of the package or method used, the Cosmos SDK supports the following metrics types:

- gauges
- summaries
- counters

Labels

Certain components of modules will have their name automatically added as a label (e.g. `BeginBlock`). Operators may also supply the application with a global set of labels that will be applied to all metrics emitted using the `telemetry` package (e.g. `chain-id`). Global labels are supplied as a list of [name, value] tuples.

Example:

```
global-labels = [
    "chain_id", "chain-OfXo4V",
```

]

Cardinality

Cardinality is key, specifically label and key cardinality. Cardinality is how many unique values of something there are. So there is naturally a tradeoff between granularity and how much stress is put on the telemetry sink in terms of indexing, scrape, and query performance.

Developers should take care to support metrics with enough dimensionality and granularity to be useful, but not increase the cardinality beyond the sink's limits. A general rule of thumb is to not exceed a cardinality of 10.

Consider the following examples with enough granularity and adequate cardinality:

- begin/end blocker time
- tx gas used
- block gas used
- amount of tokens minted
- amount of accounts created

The following examples expose too much cardinality and may not even prove to be useful:

- transfers between accounts with amount
- voting/deposit amount from unique addresses

Supported Metrics

Metric	Description	Unit	Type
tx_count	Total number of txs processed via <code>DeliverTx</code>	tx	counter
tx_successful	Total number of successful txs processed via <code>DeliverTx</code>	tx	counter
tx_failed	Total number of failed txs processed via <code>DeliverTx</code>	tx	counter
tx_gas_used	The total amount of gas used by a tx	gas	gauge
tx_gas_wanted	The total amount of gas requested by a tx	gas	gauge
tx_msg_send	The total amount of tokens sent in a <code>MsgSend</code> (per denom)	token	gauge
tx_msg_withdraw_reward	The total amount of tokens withdrawn in a <code>MsgWithdrawDelegatorReward</code> (per denom)	token	gauge
tx_msg_withdraw_commission	The total amount of tokens withdrawn in a	token	gauge

	MsgWithdrawValidatorCommission (per denom)		
tx_msg_delegate	The total amount of tokens delegated in a <code>MsgDelegate</code>	token	gauge
tx_msg_begin_unbonding	The total amount of tokens undelegated in a <code>MsgUndelegate</code>	token	gauge
tx_msg_begin_begin_redelegate	The total amount of tokens redelegated in a <code>MsgBeginRedelegate</code>	token	gauge
tx_msg_ibc_transfer	The total amount of tokens transferred via IBC in a <code>MsgTransfer</code> (source or sink chain)	token	gauge
ibc_transfer_packet_receive	The total amount of tokens received in a <code>FungibleTokenPacketData</code> (source or sink chain)	token	gauge
new_account	Total number of new accounts created	account	counter
gov_proposal	Total number of governance proposals	proposal	counter
gov_vote	Total number of governance votes for a proposal	vote	counter
gov_deposit	Total number of governance deposits for a proposal	deposit	counter
staking_delegate	Total number of delegations	delegation	counter
staking_undelegate	Total number of undegations	undelegation	counter
staking_redelegate	Total number of redelegations	redelegation	counter
ibc_transfer_send	Total number of IBC transfers sent from a chain (source or sink)	transfer	counter
ibc_transfer_receive	Total number of IBC transfers received to a chain (source or sink)	transfer	counter
ibc_client_create	Total number of clients created	create	counter
ibc_client_update	Total number of client updates	update	counter
ibc_client_upgrade	Total number of client upgrades	upgrade	counter
ibc_client_misbehaviour	Total number of client misbehaviour	misbehaviour	counter

	misbehaviours		
ibc_connection_open-init	Total number of connection OpenInit handshakes	handshake	counter
ibc_connection_open-try	Total number of connection OpenTry handshakes	handshake	counter
ibc_connection_open-ack	Total number of connection OpenAck handshakes	handshake	counter
ibc_connection_open-confirm	Total number of connection OpenConfirm handshakes	handshake	counter
ibc_channel_open-init	Total number of channel OpenInit handshakes	handshake	counter
ibc_channel_open-try	Total number of channel OpenTry handshakes	handshake	counter
ibc_channel_open-ack	Total number of channel OpenAck handshakes	handshake	counter
ibc_channel_open-confirm	Total number of channel OpenConfirm handshakes	handshake	counter
ibc_channel_close-init	Total number of channel CloseInit handshakes	handshake	counter
ibc_channel_close-confirm	Total number of channel CloseConfirm handshakes	handshake	counter
tx_msg_ibc_recv_packet	Total number of IBC packets received	packet	counter
tx_msg_ibc_acknowledge_packet	Total number of IBC packets acknowledged	acknowledgement	counter
ibc_timeout_packet	Total number of IBC timeout packets	timeout	counter
store_iavl_get	Duration of an IAVL Store#Get call	ms	summary
store_iavl_set	Duration of an IAVL Store#Set call	ms	summary
store_iavl_has	Duration of an IAVL Store#Has call	ms	summary
store_iavl_delete	Duration of an IAVL Store#Delete call	ms	summary
store_iavl_commit	Duration of an IAVL Store#Commit call	ms	summary
store_iavl_query	Duration of an IAVL Store#Query call	ms	summary

sidebar_position: 1

Object-Capability Model

Intro

When thinking about security, it is good to start with a specific threat model. Our threat model is the following:

We assume that a thriving ecosystem of Cosmos SDK modules that are easy to compose into a blockchain application will contain faulty or malicious modules.

The Cosmos SDK is designed to address this threat by being the foundation of an object capability system.

The structural properties of object capability systems favor modularity in code design and ensure reliable encapsulation in code implementation.

These structural properties facilitate the analysis of some security properties of an object-capability program or operating system. Some of these — in particular, information flow properties — can be analyzed at the level of object references and connectivity, independent of any knowledge or analysis of the code that determines the behavior of the objects.

As a consequence, these security properties can be established and maintained in the presence of new objects that contain unknown and possibly malicious code.

These structural properties stem from the two rules governing access to existing objects:

1. *An object A can send a message to B only if object A holds a reference to B.*
2. *An object A can obtain a reference to C only if object A receives a message containing a reference to C. As a consequence of these two rules, an object can obtain a reference to another object only through a preexisting chain of references. In short, "Only connectivity begets connectivity."*

For an introduction to object-capabilities, see this [Wikipedia article](#).

Ocaps in practice

The idea is to only reveal what is necessary to get the work done.

For example, the following code snippet violates the object capabilities principle:

```
type AppAccount struct {...}
account := &AppAccount{
    Address: pub.Address(),
    Coins: sdk.Coins{sdk.NewInt64Coin("ATM", 100)},
}
sumValue := externalModule.ComputeSumValue(account)
```

The method `ComputeSumValue` implies a pure function, yet the implied capability of accepting a pointer value is the capability to modify that value. The preferred method signature should take a copy instead.

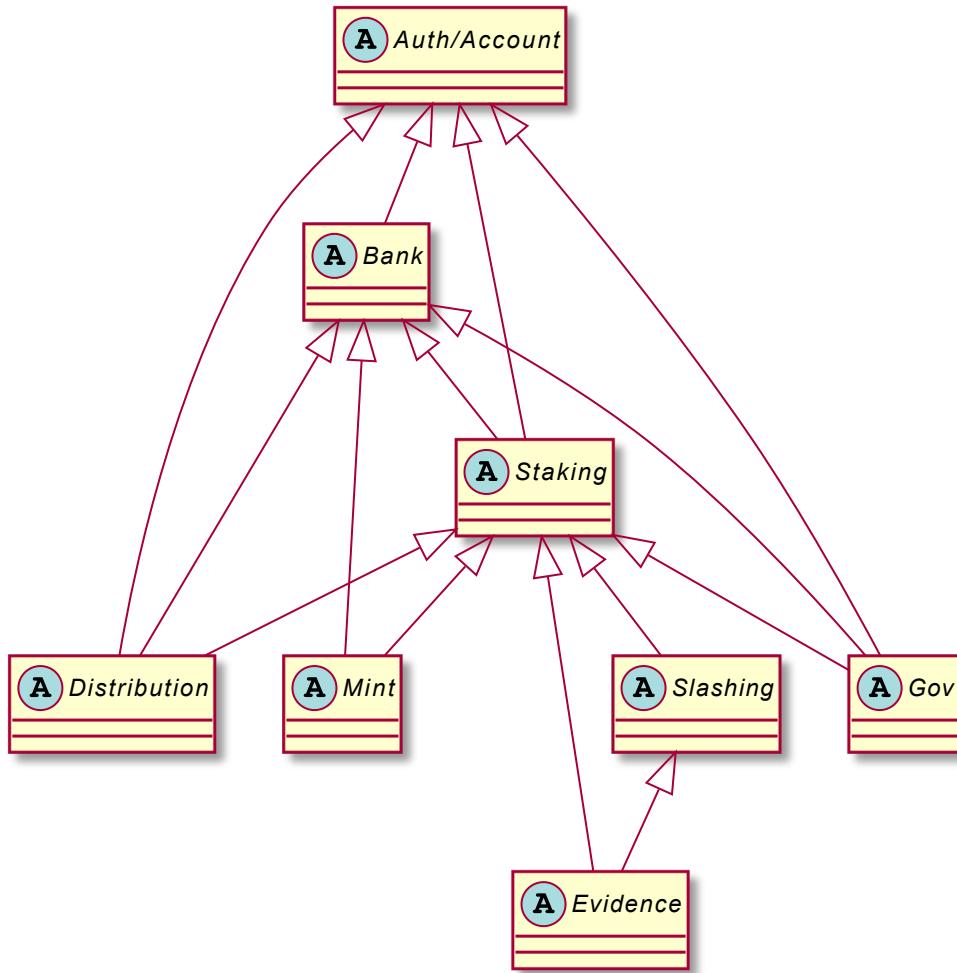
```
sumValue := externalModule.ComputeSumValue(*account)
```

In the Cosmos SDK, you can see the application of this principle in simapp.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go
```

The following diagram shows the current dependencies between keepers.

The dependencies between Keepers (Feb 2021)



sidebar_position: 1

RunTx recovery middleware

`BaseApp.runTx()` function handles Go panics that might occur during transactions execution, for example, keeper has faced an invalid state and panicked. Depending on the panic type different handler is used, for instance the default one prints an error log message. Recovery middleware is used to add custom panic recovery for Cosmos SDK application developers.

More context can be found in the corresponding [ADR-022](#) and the implementation in [recovery.go](#).

Interface

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/baseapp/recovery.go#L14-L17
```

`recoveryObj` is a return value for `recover()` function from the `builtin` Go package.

Contract:

- `RecoveryHandler` returns `nil` if `recoveryObj` wasn't handled and should be passed to the next recovery middleware;
- `RecoveryHandler` returns a non-nil `error` if `recoveryObj` was handled;

Custom RecoveryHandler register

```
BaseApp.AddRunTxRecoveryHandler(handlers ...RecoveryHandler)
```

`BaseApp` method adds recovery middleware to the default recovery chain.

Example

Lets assume we want to emit the "Consensus failure" chain state if some particular error occurred.

We have a module keeper that panics:

```
func (k FooKeeper) Do(obj interface{}) {
    if obj == nil {
        // that shouldn't happen, we need to crash the app
        err := errorsmod.Wrap(fooTypes.InternalError, "obj is nil")
        panic(err)
    }
}
```

By default that panic would be recovered and an error message will be printed to log. To override that behaviour we should register a custom `RecoveryHandler`:

```
// Cosmos SDK application constructor
customHandler := func(recoveryObj interface{}) error {
    err, ok := recoveryObj.(error)
    if !ok {
        return nil
    }

    if fooTypes.InternalError.Is(err) {
        panic(fmt.Errorf("FooKeeper did panic with error: %w", err))
    }

    return nil
}
```

```
baseApp := baseapp.NewBaseApp(....)
baseApp.AddRunTxRecoveryHandler(customHandler)
```

sidebar_position: 1

Cosmos Blockchain Simulator

The Cosmos SDK offers a full fledged simulation framework to fuzz test every message defined by a module.

On the Cosmos SDK, this functionality is provided by [SimApp](#), which is a `Baseapp` application that is used for running the [simulation](#) module. This module defines all the simulation logic as well as the operations for randomized parameters like accounts, balances etc.

Goals

The blockchain simulator tests how the blockchain application would behave under real life circumstances by generating and sending randomized messages. The goal of this is to detect and debug failures that could halt a live chain, by providing logs and statistics about the operations run by the simulator as well as exporting the latest application state when a failure was found.

Its main difference with integration testing is that the simulator app allows you to pass parameters to customize the chain that's being simulated. This comes in handy when trying to reproduce bugs that were generated in the provided operations (randomized or not).

Simulation commands

The simulation app has different commands, each of which tests a different failure type:

- `AppImportExport` : The simulator exports the initial app state and then it creates a new app with the exported `genesis.json` as an input, checking for inconsistencies between the stores.
- `AppSimulationAfterImport` : Queues two simulations together. The first one provides the app state (*i.e* genesis) to the second. Useful to test software upgrades or hard-forks from a live chain.
- `AppStateDeterminism` : Checks that all the nodes return the same values, in the same order.
- `BenchmarkInvariants` : Analysis of the performance of running all modules' invariants (*i.e* sequentially runs a [benchmark](#) test). An invariant checks for differences between the values that are on the store and the passive tracker. Eg: total coins held by accounts vs total supply tracker.
- `FullAppSimulation` : General simulation mode. Runs the chain and the specified operations for a given number of blocks. Tests that there're no `panics` on the simulation. It does also run invariant checks on every `Period` but they are not benchmarked.

Each simulation must receive a set of inputs (*i.e* flags) such as the number of blocks that the simulation is run, seed, block size, etc. Check the full list of flags [here](#).

Simulator Modes

In addition to the various inputs and commands, the simulator runs in three modes:

1. Completely random where the initial state, module parameters and simulation parameters are **pseudo-randomly generated**.
2. From a `genesis.json` file where the initial state and the module parameters are defined. This mode is helpful for running simulations on a known state such as a live network export where a new (mostly likely breaking) version of the application needs to be tested.
3. From a `params.json` file where the initial state is pseudo-randomly generated but the module and simulation parameters can be provided manually. This allows for a more controlled and deterministic simulation setup while allowing the state space to still be pseudo-randomly simulated.
The list of available parameters are listed [here](#).

:::tip These modes are not mutually exclusive. So you can for example run a randomly generated genesis state (1) with manually generated simulation params (3). :::

Usage

This is a general example of how simulations are run. For more specific examples check the Cosmos SDK [Makefile](#).

```
$ go test -mod=readonly github.com/cosmos/cosmos-sdk/simapp \
-run=TestApp<simulation_command> \
...<flags>
-v -timeout 24h
```

Debugging Tips

Here are some suggestions when encountering a simulation failure:

- Export the app state at the height where the failure was found. You can do this by passing the `-ExportStatePath` flag to the simulator.
- Use `-Verbose` logs. They could give you a better hint on all the operations involved.
- Reduce the simulation `-Period`. This will run the invariants checks more frequently.
- Print all the failed invariants at once with `-PrintAllInvariants`.
- Try using another `-Seed`. If it can reproduce the same error and if it fails sooner, you will spend less time running the simulations.
- Reduce the `-NumBlocks`. How's the app state at the height previous to the failure?
- Run invariants on every operation with `-SimulateEveryOperation`. Note: this will slow down your simulation **a lot**.
- Try adding logs to operations that are not logged. You will have to define a [Logger](#) on your `Keeper`.

Use simulation in your Cosmos SDK-based application

Learn how you can integrate the simulation into your Cosmos SDK-based application:

- Application Simulation Manager
- [Building modules: Simulator](#)
- Simulator tests

sidebar_position: 1

Protobuf Documentation

See [Cosmos SDK Buf Proto-docs](#)

sidebar_position: 1

Transaction Tips

:::note Synopsis Transaction tips are a mechanism to pay for transaction fees using another denom than the native fee denom of the chain. They are still in beta, and are not included by default in the SDK. :::

Context

In a Cosmos ecosystem where more and more chains are connected via [IBC](#), it happens that users want to perform actions on chains where they don't have native tokens yet. An example would be an Osmosis user who wants to vote on a proposal on the Cosmos Hub, but they don't have ATOMs in their wallet. A solution would be to swap OSMO for ATOM just for voting on this proposal, but that is cumbersome. Cross-chain DeFi project [Emeris](#) is another use case.

Transaction tips is a new solution for cross-chain transaction fees payment, whereby the transaction initiator signs a transaction without specifying fees, but uses a new `Tip` field. They send this signed transaction to a fee relayer who will choose the transaction fees and broadcast the final transaction, and the SDK provides a mechanism that will transfer the pre-defined `Tip` to the fee payer, to cover for fees.

Assuming we have two chains, A and B, we define the following terms:

- **the tipper:** this is the initiator of the transaction, who wants to execute a `Msg` on chain A, but doesn't have any native chain A tokens, only chain B tokens. In our example above, the tipper is the Osmosis (chain B) user wanting to vote on a Cosmos Hub (chain A) proposal.
- **the fee payer:** this is the party that will relay and broadcast the final transaction on chain A, and has chain A tokens. The tipper doesn't need to trust the feepayer.
- **the target chain:** the chain where the `Msg` is executed, chain A in this case.

Transaction Tips Flow

The transaction tips flow happens in multiple steps.

1. The tipper sends via IBC some chain B tokens to chain A. These tokens will cover for fees on the target chain A. This means that chain A's bank module holds some IBC tokens under the tipper's address.
2. The tipper drafts a transaction to be executed on the chain A. It can include chain A `Msg`s. However, instead of creating a normal transaction, they create the following `AuxSignerData` document:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L231-L244
```

where we have defined `SignDocDirectAux` as:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L68-L98
```

where `Tip` is defined as

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/proto/cosmos/tx/v1beta1/tx.proto#L233-L244
```

Notice that this document doesn't sign over the final chain A fees. Instead, it includes a `Tip` field. It also doesn't include the whole `AuthInfo` object as in `SIGN_MODE_DIRECT`, only the minimum information needed by the tipper

3. The tipper signs the `SignDocDirectAux` document and attaches the signature to the `AuxSignerData`, then sends the signed `AuxSignerData` to the fee payer.
4. From the signed `AuxSignerData` document, the fee payer constructs a transaction, using the following algorithm:
 - use as `TxBody` the exact `AuxSignerData.SignDocDirectAux.body_bytes`, to not alter the original intent of the tipper,
 - create an `AuthInfo` with:
 - `AuthInfo.Tip` copied from `AuxSignerData.SignDocDirectAux.Tip`,
 - `AuthInfo.Fee` chosen by the fee payer, which should cover for the transaction gas, but also be small enough so that the tip/fee exchange rate is economically interesting for the fee payer,
 - `AuthInfo.SignerInfos` has two signers: the first signer is the tipper, using the public key, sequence and sign mode specified in `AuxSignerData`; and the second signer is the fee payer, using their favorite sign mode,
 - a `Signatures` array with two items: the tipper's signature from `AuxSignerData.Sig`, and the final fee payer's signature.
5. Broadcast the final transaction signed by the two parties to the target chain. Once included, the Cosmos SDK will trigger a transfer of the `Tip` specified in the transaction from the tipper address to the fee payer address.

Fee Payers Market

The benefit of transaction tips for the tipper is clear: there is no need to swap tokens before executing a cross-chain message.

For the fee payer, the benefit is in the tip v.s. fee exchange. Put simply, the fee payer pays the fees of an unknown tipper's transaction, and gets in exchange the tip that the tipper chose. There is an economic incentive for the fee payer to do so only when the tip is greater than the transaction fees, given the exchange rates between the two tokens.

In the future, we imagine a market where fee payers will compete to include transactions from tippers, who on their side will optimize by specifying the lowest tip possible. A number of automated services might spin up to perform transaction gas simulation and exchange rate monitoring to optimize both the tip and fee values in real-time.

Tipper and Fee Payer Sign Modes

As we mentioned in the flow above, the tipper signs over the `SignDocDirectAux`, and the fee payer signs over the whole final transaction. As such, both parties might use different sign modes.

- The tipper MUST use `SIGN_MODE_DIRECT_AUX` or `SIGN_MODE_LEGACY_AMINO_JSON`. That is because the tipper needs to sign over the body, the tip, but not the other signers' information and not over the fee (which is unknown to the tipper).
- The fee payer MUST use `SIGN_MODE_DIRECT` or `SIGN_MODE_LEGACY_AMINO_JSON`. The fee payer signs over the whole transaction.

For example, if the fee payer signs the whole transaction with `SIGN_MODE_DIRECT_AUX`, it will be rejected by the node, as that would introduce malleability issues (`SIGN_MODE_DIRECT_AUX` doesn't sign over fees).

In both cases, using `SIGN_MODE_LEGACY_AMINO_JSON` is recommended only if hardware wallet signing is needed.

Enabling Tips on your Chain

The transaction tips functionality is introduced in Cosmos SDK v0.46, so earlier versions do not have support for tips. It is however not included by default in a v0.46 app. Sending a transaction with tips to a chain which didn't enable tips will result in a no-op, i.e. the `tip` field in the transaction will be ignored.

Enabling tips on a chain is done by adding the `TipDecorator` in the posthandler chain:

```
// HandlerOptions are the options required for constructing a SDK PostHandler which
// supports tips.
type HandlerOptions struct {
    BankKeeper types.BankKeeper
}

// MyPostHandler returns a posthandler chain with the TipDecorator.
func MyPostHandler(options HandlerOptions) (sdk.AnteHandler, error) {
    if options.BankKeeper == nil {
        return nil, errorsmod.Wrap(sdkerrors.ErrLogic, "bank keeper is required for
posthandler")
    }

    postDecorators := []sdk.AnteDecorator{
        posthandler.NewTipDecorator(options.bankKeeper),
    }

    return sdk.ChainAnteDecorators(postDecorators...), nil
}

func (app *SimApp) setPostHandler() {
    postHandler, err := MyPostHandler(
        HandlerOptions{
            BankKeeper: app.BankKeeper,
        },
    )
    if err != nil {
        panic(err)
    }
}
```

```
    app.SetPostHandler(postHandler)
}
```

Notice that `NewTipDecorator` needs a reference to the `BankKeeper`, for transferring the tip to the fee payer.

CLI Usage

The Cosmos SDK also provides some CLI tooling for the transaction tips flow, both for the tipper and for the feepayer.

For the tipper, the CLI `tx` subcommand has two new flags: `--aux` and `--tip`. The `--aux` flag is used to denote that we are creating an `AuxSignerData` instead of a `Tx`, and the `--tip` is used to populate its `Tip` field.

```
$ simd tx gov vote 16 yes --from <tipper_address> --aux --tip 50ibcdenom

### Prints the AuxSignerData as JSON:
### {"address":"cosmos1q0ayf5vq6fd2xxrwh30upg05hxdnyw2h5249a2","sign_doc":
{"body_bytes":"CosBChwvY29zbW9zLmJhbmsudjFizXRhMS5Nc2dTZW5kEmsKLWNvc21vczFxMGF5ZjV2cTzr
{@type":"/cosmos.crypto.secp256k1.PubKey","key":"AojoF/1luQ5H/nZDSrE1w3CyzGJhJdQuS7hF}
{"amount":
[{"denom":"ibcdenom","amount":"50"}],"tipper":"cosmos1q0ayf5vq6fd2xxrwh30upg05hxdnyw2h5249a2"}
```

It is useful to pipe the JSON output to a file, `> aux_signed_tx.json`

For the fee payer, the Cosmos SDK added a `tx aux-to-fee` subcommand to include an `AuxSignerData` into a transaction, add fees to it, and broadcast it.

```
$ simd tx aux-to-fee aux_signed_tx.json --from <fee_payer_address> --fees 30atom

### Prints the broadcasted tx response:
### code: 0
### codespace: sdk
### data: ""
### events: []
### gas_used: "0"
### gas_wanted: "0"
### height: "0"
### info: ""
### logs: []
### timestamp: ""
### tx: null
```

Upon completion of the second command, the fee payer's balance will be down the `30atom` fees, and up the `50ibcdenom` tip.

For both commands, the flag `--sign-mode=amino-json` is still available for hardware wallet signing.

Programmatic Usage

For the tipper, the SDK exposes a new transaction builder, the `AuxTxBuilder`, for generating an `AuxSignerData`. The API of `AuxTxBuilder` is defined [in client/tx](#), and can be used as follows:

```
// Note: there's no need to use clientCtx.TxConfig anymore.

bldr := clienttx.NewAuxTxBuilder()
err := bldr.SetMsgs(msgs...)
bldr.SetAddress("cosmos1...")
bldr.SetMemo(...)
bldr.SetTip(...)
bldr.SetPubKey(...)
err := bldr.SetSignMode(...) // DIRECT_AUX or AMINO, or else error
// ... other setters are also available

// Get the bytes to sign.
signBz, err := bldr.GetSignBytes()

// Sign the bz using your favorite method.
sig, err := privKey.sign(signBz)

// Set the signature
bldr.SetSig(sig)

// Get the final auxSignerData to be sent to the fee payer
auxSignerData, err:= bldr.GetAuxSignerData()
```

For the fee payer, the SDK added a new method on the existing `TxBuilder` to import data from an `AuxSignerData`:

```
// get `auxSignerData` from tipper, see code snippet above.

txBuilder := clientCtx.TxConfig.NewTxBuilder()
err := txBuilder.AddAuxSignerData(auxSignerData)
if err != nil {
    return err
}

// A lot of fields will be populated in txBuilder, such as its Msgs, tip
// memo, etc...

// The fee payer chooses the fee to set on the transaction.
txBuilder.SetFeePayer(<fee_payer_address>)
txBuilder.SetFeeAmount(...)
txBuilder.SetGasLimit(...)

// Usual signing code
err = authclient.SignTx(...)
if err != nil {
```

```
    return err
}
```

sidebar_position: 1

In-Place Store Migrations

:::warning Read and understand all the in-place store migration documentation before you run a migration on a live chain. :::

:::note Synopsis Upgrade your app modules smoothly with custom in-place store migration logic. :::

The Cosmos SDK uses two methods to perform upgrades:

- Exporting the entire application state to a JSON file using the `export` CLI command, making changes, and then starting a new binary with the changed JSON file as the genesis file.
- Perform upgrades in place, which significantly decrease the upgrade time for chains with a larger state. Use the [Module Upgrade Guide](#) to set up your application modules to take advantage of in-place upgrades.

This document provides steps to use the In-Place Store Migrations upgrade method.

Tracking Module Versions

Each module gets assigned a consensus version by the module developer. The consensus version serves as the breaking change version of the module. The Cosmos SDK keeps track of all module consensus versions in the `x/upgrade VersionMap` store. During an upgrade, the difference between the old `VersionMap` stored in state and the new `VersionMap` is calculated by the Cosmos SDK. For each identified difference, the module-specific migrations are run and the respective consensus version of each upgraded module is incremented.

Consensus Version

The consensus version is defined on each app module by the module developer and serves as the breaking change version of the module. The consensus version informs the Cosmos SDK on which modules need to be upgraded. For example, if the bank module was version 2 and an upgrade introduces bank module 3, the Cosmos SDK upgrades the bank module and runs the "version 2 to 3" migration script.

Version Map

The version map is a mapping of module names to consensus versions. The map is persisted to `x/upgrade`'s state for use during in-place migrations. When migrations finish, the updated version map is persisted in the state.

Upgrade Handlers

Upgrades use an `UpgradeHandler` to facilitate migrations. The `UpgradeHandler` functions implemented by the app developer must conform to the following function signature. These functions retrieve the

`VersionMap` from `x/upgrade`'s state and return the new `VersionMap` to be stored in `x/upgrade` after the upgrade. The diff between the two `VersionMap`s determines which modules need upgrading.

```
type UpgradeHandler func(ctx sdk.Context, plan Plan, fromVM VersionMap) (VersionMap, error)
```

Inside these functions, you must perform any upgrade logic to include in the provided `plan`. All upgrade handler functions must end with the following line of code:

```
return app.mm.RunMigrations(ctx, cfg, fromVM)
```

Running Migrations

Migrations are run inside of an `UpgradeHandler` using `app.mm.RunMigrations(ctx, cfg, vm)`. The `UpgradeHandler` functions describe the functionality to occur during an upgrade. The `RunMigration` function loops through the `VersionMap` argument and runs the migration scripts for all versions that are less than the versions of the new binary app module. After the migrations are finished, a new `VersionMap` is returned to persist the upgraded module versions to state.

```
cfg := module.NewConfigurator(...)  
app.UpgradeKeeper.SetUpgradeHandler("my-plan", func(ctx sdk.Context, plan  
upgradetypes.Plan, fromVM module.VersionMap) (module.VersionMap, error) {  
  
    // ...  
    // additional upgrade logic  
    // ...  
  
    // returns a VersionMap with the updated module ConsensusVersions  
    return app.mm.RunMigrations(ctx, fromVM)  
})
```

To learn more about configuring migration scripts for your modules, see the [Module Upgrade Guide](#).

Order Of Migrations

By default, all migrations are run in module name alphabetical ascending order, except `x/auth` which is run last. The reason is state dependencies between `x/auth` and other modules (you can read more in [issue #10606](#)).

If you want to change the order of migration, then you should call

`app.mm.SetOrderMigrations(module1, module2, ...)` in your `app.go` file. The function will panic if you forget to include a module in the argument list.

Adding New Modules During Upgrades

You can introduce entirely new modules to the application during an upgrade. New modules are recognized because they have not yet been registered in `x/upgrade`'s `VersionMap` store. In this case, `RunMigrations` calls the `InitGenesis` function from the corresponding module to set up its initial state.

Add StoreUpgrades for New Modules

All chains preparing to run in-place store migrations will need to manually add store upgrades for new modules and then configure the store loader to apply those upgrades. This ensures that the new module's stores are added to the multistore before the migrations begin.

```
upgradeInfo, err := app.UpgradeKeeper.ReadUpgradeInfoFromDisk()
if err != nil {
    panic(err)
}

if upgradeInfo.Name == "my-plan" &&
!app.UpgradeKeeper.IsSkipHeight(upgradeInfo.Height) {
    storeUpgrades := storetypes.StoreUpgrades{
        // add store upgrades for new modules
        // Example:
        //     Added: []string{"foo", "bar"},
        // ...
    }

    // configure store loader that checks if version == upgradeHeight and applies
    store upgrades
    app.SetStoreLoader(upgradetypes.UpgradeStoreLoader(upgradeInfo.Height,
&storeUpgrades))
}
```

Genesis State

When starting a new chain, the consensus version of each module MUST be saved to state during the application's genesis. To save the consensus version, add the following line to the `InitChainer` method in `app.go`:

```
func (app *MyApp) InitChainer(ctx sdk.Context, req abci.RequestInitChain)
abci.ResponseInitChain {
    ...
+ app.UpgradeKeeper.SetModuleVersionMap(ctx, app.mm.GetVersionMap())
    ...
}
```

This information is used by the Cosmos SDK to detect when modules with newer versions are introduced to the app.

For a new module `foo`, `InitGenesis` is called by `RunMigration` only when `foo` is registered in the module manager but it's not set in the `fromVM`. Therefore, if you want to skip `InitGenesis` when a new module is added to the app, then you should set its module version in `fromVM` to the module consensus version:

```
app.UpgradeKeeper.SetUpgradeHandler("my-plan", func(ctx sdk.Context, plan
upgradetypes.Plan, fromVM module.VersionMap) (module.VersionMap, error) {
    // ...
```

```

    // Set foo's version to the latest ConsensusVersion in the VersionMap.
    // This will skip running InitGenesis on Foo
    fromVM[foo.ModuleName] = foo.AppModule{}.ConsensusVersion()

    return app.mm.RunMigrations(ctx, fromVM)
})

```

Overwriting Genesis Functions

The Cosmos SDK offers modules that the application developer can import in their app. These modules often have an `InitGenesis` function already defined.

You can write your own `InitGenesis` function for an imported module. To do this, manually trigger your custom genesis function in the upgrade handler.

:::warning You MUST manually set the consensus version in the version map passed to the `UpgradeHandler` function. Without this, the SDK will run the Module's existing `InitGenesis` code even if you triggered your custom function in the `UpgradeHandler`. :::

```

import foo "github.com/my/module/foo"

app.UpgradeKeeper.SetUpgradeHandler("my-plan", func(ctx sdk.Context, plan
upgradetypes.Plan, fromVM module.VersionMap, error) {

    // Register the consensus version in the version map
    // to avoid the SDK from triggering the default
    // InitGenesis function.
    fromVM["foo"] = foo.AppModule{}.ConsensusVersion()

    // Run custom InitGenesis for foo
    app.mm["foo"].InitGenesis(ctx, app.appCodec, myCustomGenesisState)

    return app.mm.RunMigrations(ctx, cfg, fromVM)
})

```

Syncing a Full Node to an Upgraded Blockchain

You can sync a full node to an existing blockchain which has been upgraded using Cosmovisor

To successfully sync, you must start with the initial binary that the blockchain started with at genesis. If all Software Upgrade Plans contain binary instruction, then you can run Cosmovisor with auto-download option to automatically handle downloading and switching to the binaries associated with each sequential upgrade. Otherwise, you need to manually provide all binaries to Cosmovisor.

To learn more about Cosmovisor, see the [Cosmovisor Quick Start](#).

sidebar_position: 1

Configuration

This documentation refers to the app.toml, if you'd like to read about the config.toml please visit [CometBFT docs](#).

```
https://github.com/cosmos/cosmos-sdk/blob/main/tools/confix/data/v0.47-app.toml
```

inter-block-cache

This feature will consume more ram than a normal node, if enabled.

iavl-cache-size

Using this feature will increase ram consumption

iavl-lazy-loading

This feature is to be used for archive nodes, allowing them to have a faster start up time.

sidebar_position: 1

High-level Overview

What is the Cosmos SDK

The [Cosmos SDK](#) is an open-source framework for building multi-asset public Proof-of-Stake (PoS) blockchains, like the Cosmos Hub, as well as permissioned Proof-of-Authority (PoA) blockchains. Blockchains built with the Cosmos SDK are generally referred to as **application-specific blockchains**.

The goal of the Cosmos SDK is to allow developers to easily create custom blockchains from scratch that can natively interoperate with other blockchains. We envision the Cosmos SDK as the npm-like framework to build secure blockchain applications on top of [CometBFT](#). SDK-based blockchains are built out of composable [modules](#), most of which are open-source and readily available for any developers to use. Anyone can create a module for the Cosmos SDK, and integrating already-built modules is as simple as importing them into your blockchain application. What's more, the Cosmos SDK is a capabilities-based system that allows developers to better reason about the security of interactions between modules. For a deeper look at capabilities, jump to [Object-Capability Model](#).

What are Application-Specific Blockchains

One development paradigm in the blockchain world today is that of virtual-machine blockchains like Ethereum, where development generally revolves around building decentralized applications on top of an existing blockchain as a set of smart contracts. While smart contracts can be very good for some use cases like single-use applications (e.g. ICOs), they often fall short for building complex decentralized platforms. More generally, smart contracts can be limiting in terms of flexibility, sovereignty and performance.

Application-specific blockchains offer a radically different development paradigm than virtual-machine blockchains. An application-specific blockchain is a blockchain customized to operate a single application: developers have all the freedom to make the design decisions required for the application to run optimally. They can also provide better sovereignty, security and performance.

Learn more about [application-specific blockchains](#).

Why the Cosmos SDK

The Cosmos SDK is the most advanced framework for building custom application-specific blockchains today. Here are a few reasons why you might want to consider building your decentralized application with the Cosmos SDK:

- The default consensus engine available within the Cosmos SDK is [CometBFT](#). CometBFT is the most (and only) mature BFT consensus engine in existence. It is widely used across the industry and is considered the gold standard consensus engine for building Proof-of-Stake systems.
- The Cosmos SDK is open-source and designed to make it easy to build blockchains out of composable [modules](#). As the ecosystem of open-source Cosmos SDK modules grows, it will become increasingly easier to build complex decentralized platforms with it.
- The Cosmos SDK is inspired by capabilities-based security, and informed by years of wrestling with blockchain state-machines. This makes the Cosmos SDK a very secure environment to build blockchains.
- Most importantly, the Cosmos SDK has already been used to build many application-specific blockchains that are already in production. Among others, we can cite [Cosmos Hub](#), [IRIS Hub](#), [Binance Chain](#), [Terra](#) or [Kava](#). [Many more](#) are building on the Cosmos SDK.

Getting started with the Cosmos SDK

- Learn more about the [architecture of a Cosmos SDK application](#)
- Learn how to build an application-specific blockchain from scratch with the [Cosmos SDK Tutorial](#)

sidebar_position: 1

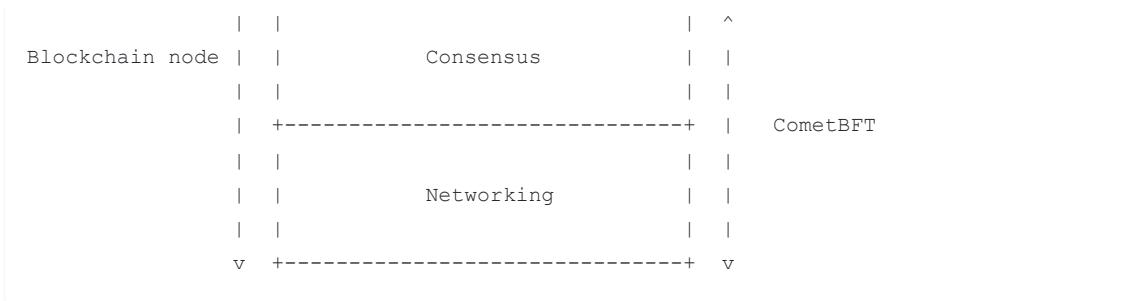
Application-Specific Blockchains

:::note Synopsis This document explains what application-specific blockchains are, and why developers would want to build one as opposed to writing Smart Contracts. :::

What are application-specific blockchains

Application-specific blockchains are blockchains customized to operate a single application. Instead of building a decentralized application on top of an underlying blockchain like Ethereum, developers build their own blockchain from the ground up. This means building a full-node client, a light-client, and all the necessary interfaces (CLI, REST, ...) to interact with the nodes.

```
^ +-----+ ^  
| | | | | | Built with Cosmos SDK  
| | State-machine = Application | |  
| | | | | v  
| +-----+ |
```



What are the shortcomings of Smart Contracts

Virtual-machine blockchains like Ethereum addressed the demand for more programmability back in 2014. At the time, the options available for building decentralized applications were quite limited. Most developers would build on top of the complex and limited Bitcoin scripting language, or fork the Bitcoin codebase which was hard to work with and customize.

Virtual-machine blockchains came in with a new value proposition. Their state-machine incorporates a virtual-machine that is able to interpret turing-complete programs called Smart Contracts. These Smart Contracts are very good for use cases like one-time events (e.g. ICOs), but they can fall short for building complex decentralized platforms. Here is why:

- Smart Contracts are generally developed with specific programming languages that can be interpreted by the underlying virtual-machine. These programming languages are often immature and inherently limited by the constraints of the virtual-machine itself. For example, the Ethereum Virtual Machine does not allow developers to implement automatic execution of code. Developers are also limited to the account-based system of the EVM, and they can only choose from a limited set of functions for their cryptographic operations. These are examples, but they hint at the lack of **flexibility** that a smart contract environment often entails.
- Smart Contracts are all run by the same virtual machine. This means that they compete for resources, which can severely restrain **performance**. And even if the state-machine were to be split in multiple subsets (e.g. via sharding), Smart Contracts would still need to be interpreted by a virtual machine, which would limit performance compared to a native application implemented at state-machine level (our benchmarks show an improvement on the order of 10x in performance when the virtual-machine is removed).
- Another issue with the fact that Smart Contracts share the same underlying environment is the resulting limitation in **sovereignty**. A decentralized application is an ecosystem that involves multiple players. If the application is built on a general-purpose virtual-machine blockchain, stakeholders have very limited sovereignty over their application, and are ultimately superseded by the governance of the underlying blockchain. If there is a bug in the application, very little can be done about it.

Application-Specific Blockchains are designed to address these shortcomings.

Application-Specific Blockchains Benefits

Flexibility

Application-specific blockchains give maximum flexibility to developers:

- In Cosmos blockchains, the state-machine is typically connected to the underlying consensus engine via an interface called the [ABCI](#). This interface can be wrapped in any programming

language, meaning developers can build their state-machine in the programming language of their choice.

- Developers can choose among multiple frameworks to build their state-machine. The most widely used today is the Cosmos SDK, but others exist (e.g. [Lotion](#), [Weave](#), ...). Typically the choice will be made based on the programming language they want to use (Cosmos SDK and Weave are in Golang, Lotion is in Javascript, ...).
- The ABCI also allows developers to swap the consensus engine of their application-specific blockchain. Today, only CometBFT is production-ready, but in the future other consensus engines are expected to emerge.
- Even when they settle for a framework and consensus engine, developers still have the freedom to tweak them if they don't perfectly match their requirements in their pristine forms.
- Developers are free to explore the full spectrum of tradeoffs (e.g. number of validators vs transaction throughput, safety vs availability in asynchrony, ...) and design choices (DB or IAVL tree for storage, UTXO or account model, ...).
- Developers can implement automatic execution of code. In the Cosmos SDK, logic can be automatically triggered at the beginning and the end of each block. They are also free to choose the cryptographic library used in their application, as opposed to being constrained by what is made available by the underlying environment in the case of virtual-machine blockchains.

The list above contains a few examples that show how much flexibility application-specific blockchains give to developers. The goal of Cosmos and the Cosmos SDK is to make developer tooling as generic and composable as possible, so that each part of the stack can be forked, tweaked and improved without losing compatibility. As the community grows, more alternatives for each of the core building blocks will emerge, giving more options to developers.

Performance

decentralized applications built with Smart Contracts are inherently capped in performance by the underlying environment. For a decentralized application to optimise performance, it needs to be built as an application-specific blockchain. Next are some of the benefits an application-specific blockchain brings in terms of performance:

- Developers of application-specific blockchains can choose to operate with a novel consensus engine such as CometBFT BFT. Compared to Proof-of-Work (used by most virtual-machine blockchains today), it offers significant gains in throughput.
- An application-specific blockchain only operates a single application, so that the application does not compete with others for computation and storage. This is the opposite of most non-sharded virtual-machine blockchains today, where smart contracts all compete for computation and storage.
- Even if a virtual-machine blockchain offered application-based sharding coupled with an efficient consensus algorithm, performance would still be limited by the virtual-machine itself. The real throughput bottleneck is the state-machine, and requiring transactions to be interpreted by a virtual-machine significantly increases the computational complexity of processing them.

Security

Security is hard to quantify, and greatly varies from platform to platform. That said here are some important benefits an application-specific blockchain can bring in terms of security:

- Developers can choose proven programming languages like Go when building their application-specific blockchains, as opposed to smart contract programming languages that are often more

immature.

- Developers are not constrained by the cryptographic functions made available by the underlying virtual-machines. They can use their own custom cryptography, and rely on well-audited crypto libraries.
- Developers do not have to worry about potential bugs or exploitable mechanisms in the underlying virtual-machine, making it easier to reason about the security of the application.

Sovereignty

One of the major benefits of application-specific blockchains is sovereignty. A decentralized application is an ecosystem that involves many actors: users, developers, third-party services, and more. When developers build on virtual-machine blockchain where many decentralized applications coexist, the community of the application is different than the community of the underlying blockchain, and the latter supersedes the former in the governance process. If there is a bug or if a new feature is needed, stakeholders of the application have very little leeway to upgrade the code. If the community of the underlying blockchain refuses to act, nothing can happen.

The fundamental issue here is that the governance of the application and the governance of the network are not aligned. This issue is solved by application-specific blockchains. Because application-specific blockchains specialize to operate a single application, stakeholders of the application have full control over the entire chain. This ensures that the community will not be stuck if a bug is discovered, and that it has the freedom to choose how it is going to evolve.

sidebar_position: 1

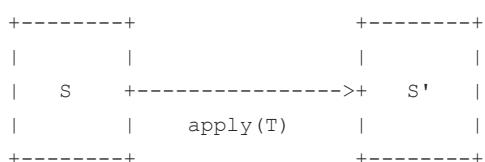
Blockchain Architecture

State machine

At its core, a blockchain is a [replicated deterministic state machine](#).

A state machine is a computer science concept whereby a machine can have multiple states, but only one at any given time. There is a `state`, which describes the current state of the system, and `transactions`, that trigger state transitions.

Given a state S and a transaction T, the state machine will return a new state S'.



In practice, the transactions are bundled in blocks to make the process more efficient. Given a state S and a block of transactions B, the state machine will return a new state S'.

```

+-----+           +-----+
|       |           | |
|   S    +-----> |   S'   |
|       |   For each T in B: apply(T) |       |
+-----+           +-----+

```

In a blockchain context, the state machine is deterministic. This means that if a node is started at a given state and replays the same sequence of transactions, it will always end up with the same final state.

The Cosmos SDK gives developers maximum flexibility to define the state of their application, transaction types and state transition functions. The process of building state-machines with the Cosmos SDK will be described more in depth in the following sections. But first, let us see how the state-machine is replicated using [CometBFT](#).

CometBFT

Thanks to the Cosmos SDK, developers just have to define the state machine, and [CometBFT](#) will handle replication over the network for them.

```

^  +-----+ ^           +-----+ ^
|  |           |           |   |   Built with Cosmos SDK
|  |   State-machine = Application |   |
|  |           |           |   v
|  +-----+           +-----+
|  |           |           |   ^
Blockchain node |  |   Consensus   |   |
|  |           |           |   |
|  +-----+           +-----+   |   CometBFT
|  |           |           |   |
|  |   Networking   |   |
|  |           |           |   |
v  +-----+           +-----+ v

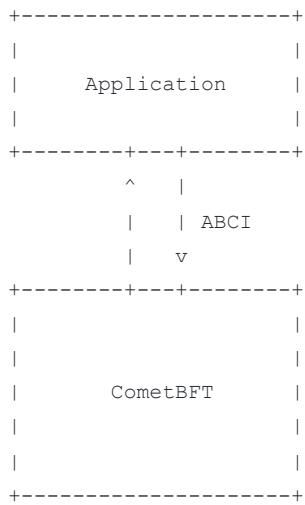
```

[CometBFT](#) is an application-agnostic engine that is responsible for handling the *networking* and *consensus* layers of a blockchain. In practice, this means that CometBFT is responsible for propagating and ordering transaction bytes. CometBFT relies on an eponymous Byzantine-Fault-Tolerant (BFT) algorithm to reach consensus on the order of transactions.

The CometBFT [consensus algorithm](#) works with a set of special nodes called *Validators*. Validators are responsible for adding blocks of transactions to the blockchain. At any given block, there is a validator set V . A validator in V is chosen by the algorithm to be the proposer of the next block. This block is considered valid if more than two thirds of V signed a `prevote` and a `precommit` on it, and if all the transactions that it contains are valid. The validator set can be changed by rules written in the state-machine.

ABCI

CometBFT passes transactions to the application through an interface called the [ABCI](#), which the application must implement.



Note that **CometBFT only handles transaction bytes**. It has no knowledge of what these bytes mean. All CometBFT does is order these transaction bytes deterministically. CometBFT passes the bytes to the application via the ABCI, and expects a return code to inform it if the messages contained in the transactions were successfully processed or not.

Here are the most important messages of the ABCI:

- `CheckTx` : When a transaction is received by CometBFT, it is passed to the application to check if a few basic requirements are met. `CheckTx` is used to protect the mempool of full-nodes against spam transactions. A special handler called the `AnteHandler` is used to execute a series of validation steps such as checking for sufficient fees and validating the signatures. If the checks are valid, the transaction is added to the `mempool` and relayed to peer nodes. Note that transactions are not processed (i.e. no modification of the state occurs) with `CheckTx` since they have not been included in a block yet.
- `DeliverTx` : When a `valid block` is received by CometBFT, each transaction in the block is passed to the application via `DeliverTx` in order to be processed. It is during this stage that the state transitions occur. The `AnteHandler` executes again, along with the actual `Msg` `service` RPC for each message in the transaction.
- `BeginBlock` / `EndBlock` : These messages are executed at the beginning and the end of each block, whether the block contains transactions or not. It is useful to trigger automatic execution of logic. Proceed with caution though, as computationally expensive loops could slow down your blockchain, or even freeze it if the loop is infinite.

Find a more detailed view of the ABCI methods from the [CometBFT docs](#).

Any application built on CometBFT needs to implement the ABCI interface in order to communicate with the underlying local CometBFT engine. Fortunately, you do not have to implement the ABCI interface. The Cosmos SDK provides a boilerplate implementation of it in the form of [baseapp](#).

sidebar_position: 1

Main Components of the Cosmos SDK

The Cosmos SDK is a framework that facilitates the development of secure state-machines on top of CometBFT. At its core, the Cosmos SDK is a boilerplate implementation of the [ABCI](#) in Golang. It comes with a [multistore](#) to persist data and a [router](#) to handle transactions.

Here is a simplified view of how transactions are handled by an application built on top of the Cosmos SDK when transferred from CometBFT via `DeliverTx`:

1. Decode `transactions` received from the CometBFT consensus engine (remember that CometBFT only deals with `[]byte`).
2. Extract `messages` from `transactions` and do basic sanity checks.
3. Route each message to the appropriate module so that it can be processed.
4. Commit state changes.

baseapp

`baseapp` is the boilerplate implementation of a Cosmos SDK application. It comes with an implementation of the ABCI to handle the connection with the underlying consensus engine. Typically, a Cosmos SDK application extends `baseapp` by embedding it in [`app.go`](#).

Here is an example of this from `simapp`, the Cosmos SDK demonstration app:

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/simapp/app.go#L170-L212
```

The goal of `baseapp` is to provide a secure interface between the store and the extensible state machine while defining as little about the state machine as possible (staying true to the ABCI).

For more on `baseapp`, please click [here](#).

Multistore

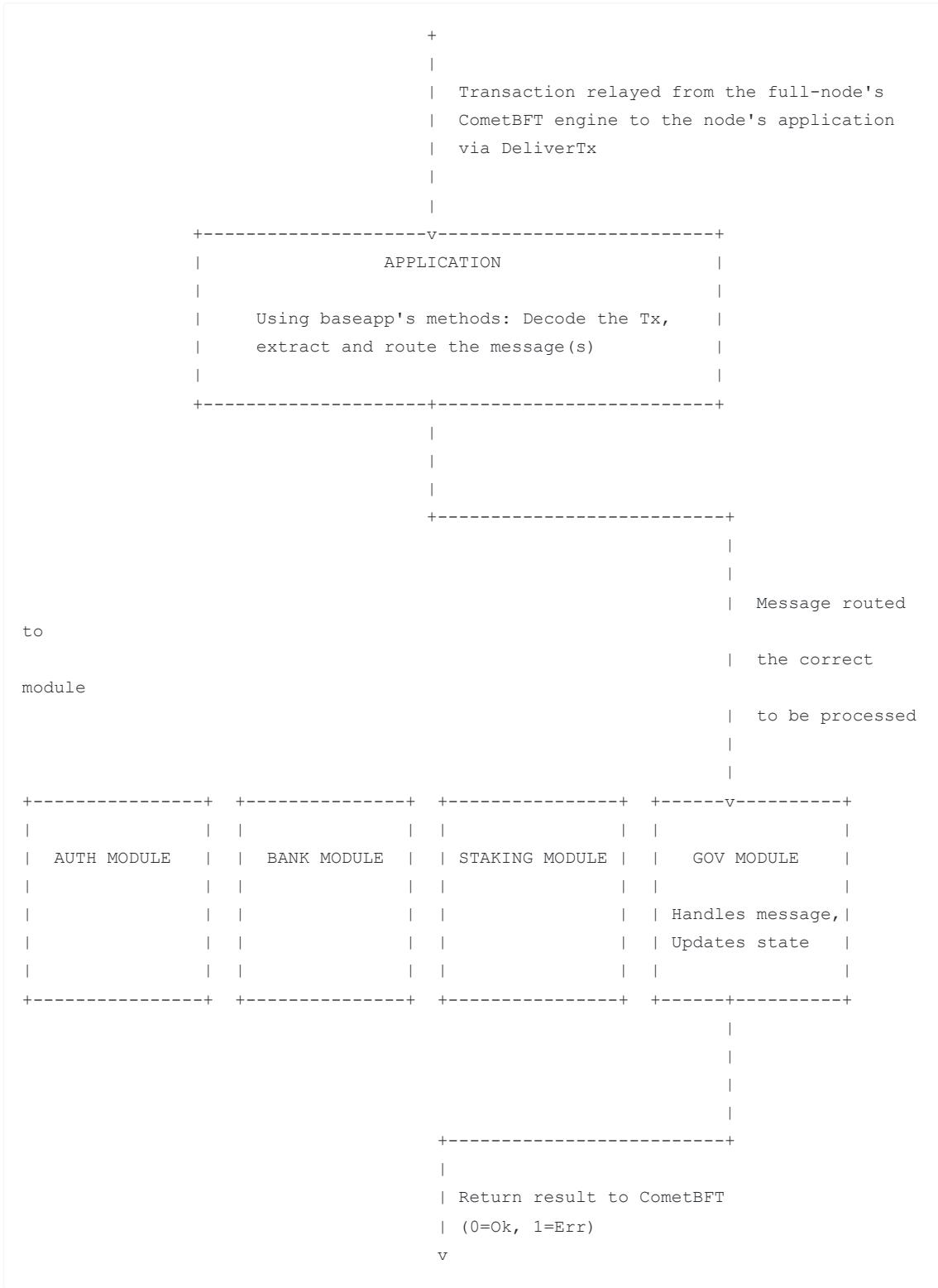
The Cosmos SDK provides a [multistore](#) for persisting state. The multistore allows developers to declare any number of [KVStores](#). These `KVStores` only accept the `[]byte` type as value and therefore any custom structure needs to be marshalled using [a codec](#) before being stored.

The multistore abstraction is used to divide the state in distinct compartments, each managed by its own module. For more on the multistore, click [here](#)

Modules

The power of the Cosmos SDK lies in its modularity. Cosmos SDK applications are built by aggregating a collection of interoperable modules. Each module defines a subset of the state and contains its own message/transaction processor, while the Cosmos SDK is responsible for routing each message to its respective module.

Here is a simplified view of how a transaction is processed by the application of each full-node when it is received in a valid block:



Each module can be seen as a little state-machine. Developers need to define the subset of the state handled by the module, as well as custom message types that modify the state (Note: messages are extracted from `transactions` by `baseapp`). In general, each module declares its own `KVStore` in the `multistore` to persist the subset of the state it defines. Most developers will need to access other 3rd party modules when building their own modules. Given that the Cosmos SDK is an open framework, some of

the modules may be malicious, which means there is a need for security principles to reason about inter-module interactions. These principles are based on [object-capabilities](#). In practice, this means that instead of having each module keep an access control list for other modules, each module implements special objects called `keepers` that can be passed to other modules to grant a pre-defined set of capabilities.

Cosmos SDK modules are defined in the `x/` folder of the Cosmos SDK. Some core modules include:

- `x/auth` : Used to manage accounts and signatures.
- `x/bank` : Used to enable tokens and token transfers.
- `x/staking + x/slashing` : Used to build Proof-Of-Stake blockchains.

In addition to the already existing modules in `x/`, that anyone can use in their app, the Cosmos SDK lets you build your own custom modules. You can check an [example of that in the tutorial](#).

sidebar_position: 1

SDK Migrations

To smoothen the update to the latest stable release, the SDK includes a CLI command for hard-fork migrations (under the `<appd> genesis migrate` subcommand). Additionally, the SDK includes in-place migrations for its core modules. These in-place migrations are useful to migrate between major releases.

- Hard-fork migrations are supported from the last major release to the current one.
- [In-place module migrations](#) are supported from the last two major releases to the current one.

Migration from a version older than the last two major releases is not supported.

When migrating from a previous version, refer to the [UPGRADING.md](#) and the [CHANGELOG.md](#) of the version you are migrating to.

sidebar_position: 0

Packages

The Cosmos SDK is a collection of Go modules. This section provides documentation on various packages that can be used when developing a Cosmos SDK chain. It lists all standalone Go modules that are part of the Cosmos SDK.

:::tip For more information on SDK modules, see the [SDK Modules](#) section. For more information on SDK tooling, see the [Tooling](#) section. :::

Core

- [Core](#) - Core library defining SDK interfaces ([ADR-063](#))
- [API](#) - API library containing generated SDK Pulsar API
- [Store](#) - Implementation of the Cosmos SDK store

State Management

- [Collections](#) - State management library
- [ORM](#) - State management library

Automation

- [Depinjekt](#) - Dependency injection framework
- [Client/v2](#) - Library powering [AutoCLI](#)

Utilities

- [Log](#) - Logging library
- [Errors](#) - Error handling library
- [Math](#) - Math library for SDK arithmetic operations

Example

- [SimApp](#) - SimApp is the sample Cosmos SDK chain. This package should not be imported in your application.
-

sidebar_position: 1

Setting up the keyring

:::note Synopsis This document describes how to configure and use the keyring and its various backends for an [application](#). :::

The keyring holds the private/public keypairs used to interact with a node. For instance, a validator key needs to be set up before running the blockchain node, so that blocks can be correctly signed. The private key can be stored in different locations, called "backends", such as a file or the operating system's own key storage.

Available backends for the keyring

Starting with the v0.38.0 release, Cosmos SDK comes with a new keyring implementation that provides a set of commands to manage cryptographic keys in a secure fashion. The new keyring supports multiple storage backends, some of which may not be available on all operating systems.

The `os` backend

The `os` backend relies on operating system-specific defaults to handle key storage securely. Typically, an operating system's credential sub-system handles password prompts, private keys storage, and user sessions according to the user's password policies. Here is a list of the most popular operating systems and their respective passwords manager:

- macOS: [Keychain](#)
- Windows: [Credentials Management API](#)
- GNU/Linux:
 - [libsecret](#)
 - [kwallet](#)

GNU/Linux distributions that use GNOME as default desktop environment typically come with [Seahorse](#). Users of KDE based distributions are commonly provided with [KDE Wallet Manager](#). Whilst the former is in fact a `libsecret` convenient frontend, the latter is a `kwallet` client.

`os` is the default option since operating system's default credentials managers are designed to meet users' most common needs and provide them with a comfortable experience without compromising on security.

The recommended backends for headless environments are `file` and `pass`.

The `file` backend

The `file` backend more closely resembles the keybase implementation used prior to v0.38.1. It stores the keyring encrypted within the app's configuration directory. This keyring will request a password each time it is accessed, which may occur multiple times in a single command resulting in repeated password prompts. If using bash scripts to execute commands using the `file` option you may want to utilize the following format for multiple prompts:

```
# assuming that KEYPASSWD is set in the environment
$ gaiacli config keyring-backend file                                # use file backend
$ (echo $KEYPASSWD; echo $KEYPASSWD) | gaiacli keys add me           # multiple prompts
$ echo $KEYPASSWD | gaiacli keys show me                            # single prompt
```

:::tip The first time you add a key to an empty keyring, you will be prompted to type the password twice. :::

The `pass` backend

The `pass` backend uses the [pass](#) utility to manage on-disk encryption of keys' sensitive data and metadata. Keys are stored inside `gpg` encrypted files within app-specific directories. `pass` is available for the most popular UNIX operating systems as well as GNU/Linux distributions. Please refer to its manual page for information on how to download and install it.

:::tip `pass` uses [GnuPG](#) for encryption. `gpg` automatically invokes the `gpg-agent` daemon upon execution, which handles the caching of GnuPG credentials. Please refer to `gpg-agent` man page for more information on how to configure cache parameters such as credentials TTL and passphrase expiration. :::

The password store must be set up prior to first use:

```
pass init <GPG_KEY_ID>
```

Replace `<GPG_KEY_ID>` with your GPG key ID. You can use your personal GPG key or an alternative one you may want to use specifically to encrypt the password store.

The `kwallet` backend

The `kwallet` backend uses [KDE Wallet Manager](#), which comes installed by default on the GNU/Linux distributions that ships KDE as default desktop environment. Please refer to [KWallet Handbook](#) for more information.

The `test` backend

The `test` backend is a password-less variation of the `file` backend. Keys are stored unencrypted on disk.

Provided for testing purposes only. The `test` backend is not recommended for use in production environments.

The `memory` backend

The `memory` backend stores keys in memory. The keys are immediately deleted after the program has exited.

Provided for testing purposes only. The `memory` backend is not recommended for use in production environments.

Setting backend using the env variable

You can set the keyring-backend using env variable: `BINNAME_KEYRING_BACKEND`. For example, if your binary name is `gaia-v5` then set: `export GAIA_V5_KEYRING_BACKEND=pass`

Adding keys to the keyring

:::warning Make sure you can build your own binary, and replace `simd` with the name of your binary in the snippets. :::

Applications developed using the Cosmos SDK come with the `keys` subcommand. For the purpose of this tutorial, we're running the `simd` CLI, which is an application built using the Cosmos SDK for testing and educational purposes. For more information, see [simapp](#).

You can use `simd keys` for help about the keys command and `simd keys [command] --help` for more information about a particular subcommand.

To create a new key in the keyring, run the `add` subcommand with a `<key_name>` argument. For the purpose of this tutorial, we will solely use the `test` backend, and call our new key `my_validator`. This key will be used in the next section.

```
$ simd keys add my_validator --keyring-backend test  
  
# Put the generated address in a variable for later use.  
MY_VALIDATOR_ADDRESS=$(simd keys show my_validator -a --keyring-backend test)
```

This command generates a new 24-word mnemonic phrase, persists it to the relevant backend, and outputs information about the keypair. If this keypair will be used to hold value-bearing tokens, be sure to write down the mnemonic phrase somewhere safe!

By default, the keyring generates a `secp256k1` keypair. The keyring also supports `ed25519` keys, which may be created by passing the `--algo ed25519` flag. A keyring can of course hold both types of keys

simultaneously, and the Cosmos SDK's `x/auth` module supports natively these two public key algorithms.

sidebar_position: 1

Running a Node

:::note Synopsis Now that the application is ready and the keyring populated, it's time to see how to run the blockchain node. In this section, the application we are running is called `simapp`, and its corresponding CLI binary `simd`. :::

:::note

Pre-requisite Readings

- [Anatomy of a Cosmos SDK Application](#)
- [Setting up the keyring](#)

:::

Initialize the Chain

:::warning Make sure you can build your own binary, and replace `simd` with the name of your binary in the snippets. :::

Before actually running the node, we need to initialize the chain, and most importantly its genesis file. This is done with the `init` subcommand:

```
# The argument <moniker> is the custom username of your node, it should be human-readable.  
simd init <moniker> --chain-id my-test-chain
```

The command above creates all the configuration files needed for your node to run, as well as a default genesis file, which defines the initial state of the network.

:::tip All these configuration files are in `~/.simapp` by default, but you can overwrite the location of this folder by passing the `--home` flag to each commands, or set an `$APPD_HOME` environment variable (where `APPD` is the name of the binary). :::

The `~/.simapp` folder has the following structure:

```
.                                         # ~/.simapp  
|- data                                     # Contains the databases used by the node.  
|- config/  
    |- app.toml                               # Application-related configuration file.  
    |- config.toml                            # CometBFT-related configuration file.  
    |- genesis.json                          # The genesis file.  
    |- node_key.json                         # Private key to use for node authentication in  
the p2p protocol.
```

```
|- priv_validator_key.json      # Private key to use as a validator in the  
consensus protocol.
```

Updating Some Default Settings

If you want to change any field values in configuration files (for ex: genesis.json) you can use `jq` ([installation](#) & [docs](#)) & `sed` commands to do that. Few examples are listed here.

```
# to change the chain-id  
jq '.chain_id = "testing"' genesis.json > temp.json && mv temp.json genesis.json  
  
# to enable the api server  
sed -i '/\[api\]/,+3 s/enable = false/enable = true/' app.toml  
  
# to change the voting_period  
jq '.app_state.gov.voting_params.voting_period = "600s"' genesis.json > temp.json &&  
mv temp.json genesis.json  
  
# to change the inflation  
jq '.app_state.mint.minter.inflation = "0.30000000000000000000"' genesis.json >  
temp.json && mv temp.json genesis.json
```

Client Interaction

When instantiating a node, GRPC and REST are defaulted to localhost to avoid unknown exposure of your node to the public. It is recommended to not expose these endpoints without a proxy that can handle load balancing or authentication is setup between your node and the public.

:::tip A commonly used tool for this is [nginx](#). :::

Adding Genesis Accounts

Before starting the chain, you need to populate the state with at least one account. To do so, first [create a new account in the keyring](#) named `my_validator` under the `test` keyring backend (feel free to choose another name and another backend).

Now that you have created a local account, go ahead and grant it some `stake` tokens in your chain's genesis file. Doing so will also make sure your chain is aware of this account's existence:

```
simd genesis add-genesis-account $MY_VALIDATOR_ADDRESS 100000000000stake
```

Recall that `$MY_VALIDATOR_ADDRESS` is a variable that holds the address of the `my_validator` key in the [keyring](#). Also note that the tokens in the Cosmos SDK have the `{amount}{denom}` format: `amount` is a 18-digit-precision decimal number, and `denom` is the unique token identifier with its denomination key (e.g. `atom` or `uatom`). Here, we are granting `stake` tokens, as `stake` is the token identifier used for staking in [simapp](#). For your own chain with its own staking denom, that token identifier should be used instead.

Now that your account has some tokens, you need to add a validator to your chain. Validators are special full-nodes that participate in the consensus process (implemented in the [underlying consensus engine](#)) in order to add new blocks to the chain. Any account can declare its intention to become a validator operator,

but only those with sufficient delegation get to enter the active set (for example, only the top 125 validator candidates with the most delegation get to be validators in the Cosmos Hub). For this guide, you will add your local node (created via the `init` command above) as a validator of your chain. Validators can be declared before a chain is first started via a special transaction included in the genesis file called a `gentx`:

```
# Create a gentx.  
simd genesis gentx my_validator 100000000stake --chain-id my-test-chain --keyring-backend test  
  
# Add the gentx to the genesis file.  
simd genesis collect-gentxs
```

A `gentx` does three things:

1. Registers the `validator` account you created as a validator operator account (i.e. the account that controls the validator).
2. Self-delegates the provided `amount` of staking tokens.
3. Link the operator account with a CometBFT node pubkey that will be used for signing blocks. If no `--pubkey` flag is provided, it defaults to the local node pubkey created via the `simd init` command above.

For more information on `gentx`, use the following command:

```
simd genesis gentx --help
```

Configuring the Node Using `app.toml` and `config.toml`

The Cosmos SDK automatically generates two configuration files inside `~/.simapp/config`:

- `config.toml` : used to configure the CometBFT, learn more on [CometBFT's documentation](#),
- `app.toml` : generated by the Cosmos SDK, and used to configure your app, such as state pruning strategies, telemetry, gRPC and REST servers configuration, state sync...

Both files are heavily commented, please refer to them directly to tweak your node.

One example config to tweak is the `minimum-gas-prices` field inside `app.toml`, which defines the minimum gas prices the validator node is willing to accept for processing a transaction. Depending on the chain, it might be an empty string or not. If it's empty, make sure to edit the field with some value, for example `10token`, or else the node will halt on startup. For the purpose of this tutorial, let's set the minimum gas price to 0:

```
# The minimum gas prices a validator is willing to accept for processing a  
# transaction. A transaction's fees must meet the minimum of any denomination  
# specified in this config (e.g. 0.25token1;0.0001token2).  
minimum-gas-prices = "0stake"
```

:::tip When running a node (not a validator!) and not wanting to run the application mempool, set the `max-txs` field to `-1`.

```
[mempool]
# Setting max-txs to 0 will allow for an unbounded amount of transactions in the mempool.
# Setting max_txs to negative 1 (-1) will disable transactions from being inserted into the mempool.
# Setting max_txs to a positive number (> 0) will limit the number of transactions in the mempool, by the specified amount.
#
# Note, this configuration only applies to SDK built-in app-side mempool
# implementations.
max-txs = "-1"
```

⋮

Run a Localnet

Now that everything is set up, you can finally start your node:

```
simd start
```

You should see blocks come in.

The previous command allows you to run a single node. This is enough for the next section on interacting with this node, but you may wish to run multiple nodes at the same time, and see how consensus happens between them.

The naive way would be to run the same commands again in separate terminal windows. This is possible, however in the Cosmos SDK, we leverage the power of [Docker Compose](#) to run a localnet. If you need inspiration on how to set up your own localnet with Docker Compose, you can have a look at the Cosmos SDK's [docker-compose.yml](#).

Logging

Logging provides a way to see what is going on with a node. By default the info level is set. This is a global level and all info logs will be outputted to the terminal. If you would like to filter specific logs to the terminal instead of all, then setting `module:log_level` is how this can work.

Example:

In config.toml:

```
log_level: "state:info,p2p:info,consensus:info,x/staking:info,x/ibc:info,*error"
```

State Sync

State sync is the act in which a node syncs the latest or close to the latest state of a blockchain. This is useful for users who don't want to sync all the blocks in history. Read more in [CometBFT documentation](#).

State sync works thanks to snapshots. Read how the SDK handles snapshots [here](#).

Local State Sync

Local state sync work similar to normal state sync except that it works off a local snapshot of state instead of one provided via the p2p network. The steps to start local state sync are similar to normal state sync with a few different designs.

1. As mentioned in <https://docs.cometbft.com/v0.37/core/state-sync>, one must set a height and hash in the config.toml along with a few rpc servers (the afromentioned link has instructions on how to do this).
2. Run `<appd snapshot restore <height> <format>` to restore a local snapshot (note: first load it from a file with the `load` command).
3. Bootsraping Comet state in order to start the node after the snapshot has been ingested. This can be done with the bootstrap command `<app> comet bootstrap-state`

Snapshots Commands

The Cosmos SDK provides commands for managing snapshots. These commands can be added in an app with the following snippet in `cmd/<app>/root.go` :

```
import (
    "github.com/cosmos/cosmos-sdk/client/snapshot"
)

func initRootCmd(/* ... */) {
    // ...
    rootCmd.AddCommand(
        snapshot.Cmd(appCreator),
    )
}
```

Then following commands are available at `<appd> snapshots [command]` :

- **list**: list local snapshots
- **load**: Load a snapshot archive file into snapshot store
- **restore**: Restore app state from local snapshot
- **export**: Export app state to snapshot store
- **dump**: Dump the snapshot as portable archive format
- **delete**: Delete a local snapshot

sidebar_position: 1

Interacting with the Node

:::note Synopsis There are multiple ways to interact with a node: using the CLI, using gRPC or using the REST endpoints. :::

:::note

Pre-requisite Readings

- [gRPC, REST and CometBFT Endpoints](#)

- [Running a Node](#)

⋮

Using the CLI

Now that your chain is running, it is time to try sending tokens from the first account you created to a second account. In a new terminal window, start by running the following query command:

```
simd query bank balances $MY_VALIDATOR_ADDRESS
```

You should see the current balance of the account you created, equal to the original balance of `stake` you granted it minus the amount you delegated via the `gentx`. Now, create a second account:

```
simd keys add recipient --keyring-backend test

# Put the generated address in a variable for later use.
RECIPIENT=$(simd keys show recipient -a --keyring-backend test)
```

The command above creates a local key-pair that is not yet registered on the chain. An account is created the first time it receives tokens from another account. Now, run the following command to send tokens to the `recipient` account:

```
simd tx bank send $MY_VALIDATOR_ADDRESS $RECIPIENT 1000000stake --chain-id my-test-chain --keyring-backend test

# Check that the recipient account did receive the tokens.
simd query bank balances $RECIPIENT
```

Finally, delegate some of the stake tokens sent to the `recipient` account to the validator:

```
simd tx staking delegate $(simd keys show my_validator --bech val -a --keyring-backend test) 500stake --from recipient --chain-id my-test-chain --keyring-backend test

# Query the total delegations to `validator`.
simd query staking delegations-to $(simd keys show my_validator --bech val -a --keyring-backend test)
```

You should see two delegations, the first one made from the `gentx`, and the second one you just performed from the `recipient` account.

Using gRPC

The Protobuf ecosystem developed tools for different use cases, including code-generation from `*.proto` files into various languages. These tools allow the building of clients easily. Often, the client connection (i.e. the transport) can be plugged and replaced very easily. Let's explore one of the most popular transport: [gRPC](#).

Since the code generation library largely depends on your own tech stack, we will only present three alternatives:

- `grpcurl` for generic debugging and testing,
- programmatically via Go,
- CosmJS for JavaScript/TypeScript developers.

grpcurl

`grpcurl` is like `curl` but for gRPC. It is also available as a Go library, but we will use it only as a CLI command for debugging and testing purposes. Follow the instructions in the previous link to install it.

Assuming you have a local node running (either a localnet, or connected a live network), you should be able to run the following command to list the Protobuf services available (you can replace `localhost:9000` by the gRPC server endpoint of another node, which is configured under the `grpc.address` field inside `app.toml`):

```
grpcurl -plaintext localhost:9090 list
```

You should see a list of gRPC services, like `cosmos.bank.v1beta1.Query`. This is called reflection, which is a Protobuf endpoint returning a description of all available endpoints. Each of these represents a different Protobuf service, and each service exposes multiple RPC methods you can query against.

In order to get a description of the service you can run the following command:

```
grpcurl -plaintext \
localhost:9090 \
describe cosmos.bank.v1beta1.Query           # Service we want to inspect
```

It's also possible to execute an RPC call to query the node for information:

```
grpcurl \
-plaintext \
-d "{\"address\":\"$MY_VALIDATOR_ADDRESS\"} \
localhost:9090 \
cosmos.bank.v1beta1.Query/AllBalances
```

The list of all available gRPC query endpoints is [coming soon](#).

Query for historical state using grpcurl

You may also query for historical data by passing some [gRPC metadata](#) to the query: the `x-cosmos-block-height` metadata should contain the block to query. Using `grpcurl` as above, the command looks like:

```
grpcurl \
-plaintext \
-H "x-cosmos-block-height: 123" \
-d "{\"address\":\"$MY_VALIDATOR_ADDRESS\"} \
localhost:9090 \
cosmos.bank.v1beta1.Query/AllBalances
```

Assuming the state at that block has not yet been pruned by the node, this query should return a non-empty response.

Programmatically via Go

The following snippet shows how to query the state using gRPC inside a Go program. The idea is to create a gRPC connection, and use the Protobuf-generated client code to query the gRPC server.

Install Cosmos SDK

```
go get github.com/cosmos/cosmos-sdk@main
```

```
package main

import (
    "context"
    "fmt"

    "google.golang.org/grpc"

    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    banktypes "github.com/cosmos/cosmos-sdk/x/bank/types"
)

func queryState() error {
    myAddress, err := sdk.AccAddressFromBech32("cosmos1...") // the my_validator or
    recipient address.
    if err != nil {
        return err
    }

    // Create a connection to the gRPC server.
    grpcConn, err := grpc.Dial(
        "127.0.0.1:9090", // your gRPC server address.
        grpc.WithInsecure(), // The Cosmos SDK doesn't support any transport
    security mechanism.

        // This instantiates a general gRPC codec which handles proto bytes. We pass
    in a nil interface registry
        // if the request/response types contain interface instead of 'nil' you
    should pass the application specific codec.

    grpc.WithDefaultCallOptions(grpc.ForceCodec(codec.NewProtoCodec(nil).GRPCCodec())),
    )
    if err != nil {
        return err
    }
    defer grpcConn.Close()

    // This creates a gRPC client to query the x/bank service.
    bankClient := banktypes.NewQueryClient(grpcConn)
```

```

bankRes, err := bankClient.Balance(
    context.Background(),
    &banktypes.QueryBalanceRequest{Address: myAddress.String(), Denom: "stake"},
)
if err != nil {
    return err
}

fmt.Println(bankRes.GetBalance()) // Prints the account balance

return nil
}

func main() {
    if err := queryState(); err != nil {
        panic(err)
    }
}

```

You can replace the query client (here we are using `x/bank`'s) with one generated from any other Protobuf service. The list of all available gRPC query endpoints is [coming soon](#).

Query for historical state using Go

Querying for historical blocks is done by adding the block height metadata in the gRPC request.

```

package main

import (
    "context"
    "fmt"

    "google.golang.org/grpc"
    "google.golang.org/grpc/metadata"

    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    grpctypes "github.com/cosmos/cosmos-sdk/types/grpc"
    banktypes "github.com/cosmos/cosmos-sdk/x/bank/types"
)

func queryState() error {
    myAddress, err :=
    sdk.AccAddressFromBech32("cosmos1yerherx4d43gj5wa3z15vflj9d4pln42n7kuzu") // the
    my_validator or recipient address.
    if err != nil {
        return err
    }

    // Create a connection to the gRPC server.
    grpcConn, err := grpc.Dial(
        "127.0.0.1:9090", // your gRPC server address.

```

```

    grpc.WithInsecure(), // The Cosmos SDK doesn't support any transport
    security mechanism.

    // This instantiates a general gRPC codec which handles proto bytes. We pass
    in a nil interface registry

    // if the request/response types contain interface instead of 'nil' you
    should pass the application specific codec.

grpc.WithDefaultCallOptions(grpc.ForceCodec(codec.NewProtoCodec(nil).GRPCCodec())),
)
if err != nil {
    return err
}
defer grpcConn.Close()

// This creates a gRPC client to query the x/bank service.
bankClient := banktypes.NewQueryClient(grpcConn)

var header metadata.MD
_, err = bankClient.Balance(
    metadata.AppendToOutgoingContext(context.Background(),
        grpctypes.GRPCBlockHeightHeader, "12"), // Add metadata to request
    &banktypes.QueryBalanceRequest{Address: myAddress.String(), Denom: "stake"},
    grpc.Header(&header), // Retrieve header from response
)
if err != nil {
    return err
}
blockHeight := header.Get(grpctypes.GRPCBlockHeightHeader)

fmt.Println(blockHeight) // Prints the block height (12)

return nil
}

func main() {
    if err := queryState(); err != nil {
        panic(err)
    }
}

```

CosmJS

CosmJS documentation can be found at <https://cosmos.github.io/cosmjs>. As of January 2021, CosmJS documentation is still work in progress.

Using the REST Endpoints

As described in the [gRPC guide](#), all gRPC services on the Cosmos SDK are made available for more convenient REST-based queries through gRPC-gateway. The format of the URL path is based on the Protobuf service method's full-qualified name, but may contain small customizations so that final URLs look more idiomatic. For example, the REST endpoint for the `cosmos.bank.v1beta1.Query/AllBalances`

method is `GET /cosmos/bank/v1beta1/balances/{address}`. Request arguments are passed as query parameters.

Note that the REST endpoints are not enabled by default. To enable them, edit the `api` section of your `~/.simapp/config/app.toml` file:

```
# Enable defines if the API server should be enabled.  
enable = true
```

As a concrete example, the `curl` command to make balances request is:

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
http://localhost:1317/cosmos/bank/v1beta1/balances/$MY_VALIDATOR_ADDRESS
```

Make sure to replace `localhost:1317` with the REST endpoint of your node, configured under the `api.address` field.

The list of all available REST endpoints is available as a Swagger specification file, it can be viewed at `localhost:1317/swagger`. Make sure that the `api.swagger` field is set to true in your `app.toml` file.

Query for historical state using REST

Querying for historical state is done using the HTTP header `x-cosmos-block-height`. For example, a curl command would look like:

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
-H "x-cosmos-block-height: 123" \  
http://localhost:1317/cosmos/bank/v1beta1/balances/$MY_VALIDATOR_ADDRESS
```

Assuming the state at that block has not yet been pruned by the node, this query should return a non-empty response.

Cross-Origin Resource Sharing (CORS)

CORS policies are not enabled by default to help with security. If you would like to use the rest-server in a public environment we recommend you provide a reverse proxy, this can be done with [nginx](#). For testing and development purposes there is an `enabled-unsafe-cors` field inside `app.toml`.

sidebar_position: 1

Generating, Signing and Broadcasting Transactions

:::note Synopsis This document describes how to generate an (unsigned) transaction, signing it (with one or multiple keys), and broadcasting it to the network. :::

Using the CLI

The easiest way to send transactions is using the CLI, as we have seen in the previous page when [interacting with a node](#). For example, running the following command

```
simd tx bank send $MY_VALIDATOR_ADDRESS $RECIPIENT 1000stake --chain-id my-test-chain --keyring-backend test
```

will run the following steps:

- generate a transaction with one `Msg` (`x/bank`'s `MsgSend`), and print the generated transaction to the console.
- ask the user for confirmation to send the transaction from the `$MY_VALIDATOR_ADDRESS` account.
- fetch `$MY_VALIDATOR_ADDRESS` from the keyring. This is possible because we have [set up the CLI's keyring](#) in a previous step.
- sign the generated transaction with the keyring's account.
- broadcast the signed transaction to the network. This is possible because the CLI connects to the node's CometBFT RPC endpoint.

The CLI bundles all the necessary steps into a simple-to-use user experience. However, it's possible to run all the steps individually too.

Generating a Transaction

Generating a transaction can simply be done by appending the `--generate-only` flag on any `tx` command, e.g.:

```
simd tx bank send $MY_VALIDATOR_ADDRESS $RECIPIENT 1000stake --chain-id my-test-chain --generate-only
```

This will output the unsigned transaction as JSON in the console. We can also save the unsigned transaction to a file (to be passed around between signers more easily) by appending `> unsigned_tx.json` to the above command.

Signing a Transaction

Signing a transaction using the CLI requires the unsigned transaction to be saved in a file. Let's assume the unsigned transaction is in a file called `unsigned_tx.json` in the current directory (see previous paragraph on how to do that). Then, simply run the following command:

```
simd tx sign unsigned_tx.json --chain-id my-test-chain --keyring-backend test --from $MY_VALIDATOR_ADDRESS
```

This command will decode the unsigned transaction and sign it with `SIGN_MODE_DIRECT` with `$MY_VALIDATOR_ADDRESS`'s key, which we already set up in the keyring. The signed transaction will be output as JSON to the console, and, as above, we can save it to a file by appending `--output-document` `signed_tx.json`.

Some useful flags to consider in the `tx sign` command:

- `--sign-mode` : you may use `amino-json` to sign the transaction using `SIGN_MODE_LEGACY_AMINO_JSON`,
- `--offline` : sign in offline mode. This means that the `tx sign` command doesn't connect to the node to retrieve the signer's account number and sequence, both needed for signing. In this case, you must manually supply the `--account-number` and `--sequence` flags. This is useful for offline signing, i.e. signing in a secure environment which doesn't have access to the internet.

Siging with Multiple Signers

:::warning Please note that signing a transaction with multiple signers or with a multisig account, where at least one signer uses `SIGN_MODE_DIRECT`, is not yet possible. You may follow [this Github issue](#) for more info. :::

Siging with multiple signers is done with the `tx multisign` command. This command assumes that all signers use `SIGN_MODE_LEGACY_AMINO_JSON`. The flow is similar to the `tx sign` command flow, but instead of signing an unsigned transaction file, each signer signs the file signed by previous signer(s). The `tx multisign` command will append signatures to the existing transactions. It is important that signers sign the transaction **in the same order** as given by the transaction, which is retrievable using the `GetSigners()` method.

For example, starting with the `unsigned_tx.json`, and assuming the transaction has 4 signers, we would run:

```
# Let signer1 sign the unsigned tx.  
simd tx multisign unsigned_tx.json signer_key_1 --chain-id my-test-chain --keyring-backend test > partial_tx_1.json  
# Now signer1 will send the partial_tx_1.json to the signer2.  
# Signer2 appends their signature:  
simd tx multisign partial_tx_1.json signer_key_2 --chain-id my-test-chain --keyring-backend test > partial_tx_2.json  
# Signer2 sends the partial_tx_2.json file to signer3, and signer3 can append his signature:  
simd tx multisign partial_tx_2.json signer_key_3 --chain-id my-test-chain --keyring-backend test > partial_tx_3.json
```

Broadcasting a Transaction

Broadcasting a transaction is done using the following command:

```
simd tx broadcast tx_signed.json
```

You may optionally pass the `--broadcast-mode` flag to specify which response to receive from the node:

- `sync` : the CLI waits for a CheckTx execution response only.

- `async` : the CLI returns immediately (transaction might fail).

Encoding a Transaction

In order to broadcast a transaction using the gRPC or REST endpoints, the transaction will need to be encoded first. This can be done using the CLI.

Encoding a transaction is done using the following command:

```
simd tx encode tx_signed.json
```

This will read the transaction from the file, serialize it using Protobuf, and output the transaction bytes as base64 in the console.

Decoding a Transaction

The CLI can also be used to decode transaction bytes.

Decoding a transaction is done using the following command:

```
simd tx decode [protobuf-byte-string]
```

This will decode the transaction bytes and output the transaction as JSON in the console. You can also save the transaction to a file by appending `> tx.json` to the above command.

Programmatically with Go

It is possible to manipulate transactions programmatically via Go using the Cosmos SDK's `TxBuilder` interface.

Generating a Transaction

Before generating a transaction, a new instance of a `TxBuilder` needs to be created. Since the Cosmos SDK supports both Amino and Protobuf transactions, the first step would be to decide which encoding scheme to use. All the subsequent steps remain unchanged, whether you're using Amino or Protobuf, as `TxBuilder` abstracts the encoding mechanisms. In the following snippet, we will use Protobuf.

```
import (
    "github.com/cosmos/cosmos-sdk/simapp"
)

func sendTx() error {
    // Choose your codec: Amino or Protobuf. Here, we use Protobuf, given by the
    // following function.
    app := simapp.NewSimApp(...)

    // Create a new TxBuilder.
    txBuilder := app.TxConfig().NewTxBuilder()

    // --snip--
}
```

We can also set up some keys and addresses that will send and receive the transactions. Here, for the purpose of the tutorial, we will be using some dummy data to create keys.

```
import (
    "github.com/cosmos/cosmos-sdk/testutil/testdata"
)

priv1, _, addr1 := testdata.KeyTestPubAddr()
priv2, _, addr2 := testdata.KeyTestPubAddr()
priv3, _, addr3 := testdata.KeyTestPubAddr()
```

Populating the `TxBuilder` can be done via its methods:

https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/client/tx_config.go#L33-L50

```
import (
    banktypes "github.com/cosmos/cosmos-sdk/x/bank/types"
)

func sendTx() error {
    // --snip--

    // Define two x/bank MsgSend messages:
    // - from addr1 to addr3,
    // - from addr2 to addr3.
    // This means that the transactions needs two signers: addr1 and addr2.
    msg1 := banktypes.NewMsgSend(addr1, addr3,
        types.NewCoins(types.NewInt64Coin("atom", 12)))
    msg2 := banktypes.NewMsgSend(addr2, addr3,
        types.NewCoins(types.NewInt64Coin("atom", 34)))

    err := txBuilder.SetMsgs(msg1, msg2)
    if err != nil {
        return err
    }

    txBuilder.SetGasLimit(...)
    txBuilder.SetFeeAmount(...)
    txBuilder.SetMemo(...)
    txBuilder.setTimeoutHeight(...)
}
```

At this point, `TxBuilder`'s underlying transaction is ready to be signed.

Siging a Transaction

We set encoding config to use Protobuf, which will use `SIGN_MODE_DIRECT` by default. As per [ADR-020](#), each signer needs to sign the `SignerInfo`s of all other signers. This means that we need to perform two steps sequentially:

- for each signer, populate the signer's `SignerInfo` inside `TxBuilder`,
- once all `SignerInfo`s are populated, for each signer, sign the `SignDoc` (the payload to be signed).

In the current `TxBuilder`'s API, both steps are done using the same method: `SetSignatures()`. The current API requires us to first perform a round of `SetSignatures()` *with empty signatures*, only to populate `SignerInfo`s, and a second round of `SetSignatures()` to actually sign the correct payload.

```

import (
    cryptotypes "github.com/cosmos/cosmos-sdk/crypto/types"
    "github.com/cosmos/cosmos-sdk/types/tx/signing"
    xauthsigning "github.com/cosmos/cosmos-sdk/x/auth/signing"
)

func sendTx() error {
    // --snip--

    privs := []cryptotypes.PrivKey{priv1, priv2}
    accNums:= []uint64{..., ...} // The accounts' account numbers
    accSeqs:= []uint64{..., ...} // The accounts' sequence numbers

    // First round: we gather all the signer infos. We use the "set empty
    // signature" hack to do that.
    var sigsV2 []signing.SignatureV2
    for i, priv := range privs {
        sigV2 := signing.SignatureV2{
            PubKey: priv.PubKey(),
            Data: &signing.SingleSignatureData{
                SignMode: encCfg.TxConfig.SignModeHandler().DefaultMode(),
                Signature: nil,
            },
            Sequence: accSeqs[i],
        }

        sigsV2 = append(sigsV2, sigV2)
    }
    err := txBuilder.SetSignatures(sigsV2...)
    if err != nil {
        return err
    }

    // Second round: all signer infos are set, so each signer can sign.
    sigsV2 = []signing.SignatureV2{}
    for i, priv := range privs {
        signerData := xauthsigning.SignerData{
            ChainID:      chainID,
            AccountNumber: accNums[i],
            Sequence:     accSeqs[i],
        }
        sigV2, err := tx.SignWithPrivateKey(
            encCfg.TxConfig.SignModeHandler().DefaultMode(), signerData,

```

```

        txBuilder, priv, encCfg.TxConfig, accSeqs[i])
    if err != nil {
        return nil, err
    }

    sigsV2 = append(sigsV2, sigV2)
}
err = txBuilder.SetSignatures(sigsV2...)
if err != nil {
    return err
}
}

```

The `TxBuilder` is now correctly populated. To print it, you can use the `TxConfig` interface from the initial encoding config `encCfg`:

```

func sendTx() error {
// --snip--

// Generated Protobuf-encoded bytes.
txBytes, err := encCfg.TxConfig.TxEncoder()(txBuilder.GetTx())
if err != nil {
    return err
}

// Generate a JSON string.
txJSONBytes, err := encCfg.TxConfig.TxJSONEncoder()(txBuilder.GetTx())
if err != nil {
    return err
}
txJSON := string(txJSONBytes)
}

```

Broadcasting a Transaction

The preferred way to broadcast a transaction is to use gRPC, though using REST (via `gRPC-gateway`) or the CometBFT RPC is also possible. An overview of the differences between these methods is exposed [here](#). For this tutorial, we will only describe the gRPC method.

```

import (
    "context"
    "fmt"

    "google.golang.org/grpc"

    "github.com/cosmos/cosmos-sdk/types/tx"
)

func sendTx(ctx context.Context) error {
// --snip--

```

```

// Create a connection to the gRPC server.
grpcConn := grpc.Dial(
    "127.0.0.1:9090", // Or your gRPC server address.
    grpc.WithInsecure(), // The Cosmos SDK doesn't support any transport
    security mechanism.
)
defer grpcConn.Close()

// Broadcast the tx via gRPC. We create a new client for the Protobuf Tx
// service.
txClient := tx.NewServiceClient(grpcConn)
// We then call the BroadcastTx method on this client.
grpcRes, err := txClient.BroadcastTx(
    ctx,
    &tx.BroadcastTxRequest{
        Mode:    tx.BroadcastMode_BROADCAST_MODE_SYNC,
        TxBytes: txBytes, // Proto-binary of the signed transaction, see
previous step.
    },
)
if err != nil {
    return err
}

fmt.Println(grpcRes.TxResponse.Code) // Should be `0` if the tx is successful

return nil
}

```

Simulating a Transaction

Before broadcasting a transaction, we sometimes may want to dry-run the transaction, to estimate some information about the transaction without actually committing it. This is called simulating a transaction, and can be done as follows:

```

import (
    "context"
    "fmt"
    "testing"

    "github.com/cosmos/cosmos-sdk/client"
    "github.com/cosmos/cosmos-sdk/types/tx"
    authTx "github.com/cosmos/cosmos-sdk/x/auth/tx"
)

func simulateTx() error {
    // --snip--

    // Simulate the tx via gRPC. We create a new client for the Protobuf Tx
    // service.
}

```

```

txClient := tx.NewServiceClient(grpcConn)
txBytes := /* Fill in with your signed transaction bytes. */

// We then call the Simulate method on this client.
grpcRes, err := txClient.Simulate(
    context.Background(),
    &tx.SimulateRequest{
        TxBytes: txBytes,
    },
)
if err != nil {
    return err
}

fmt.Println(grpcRes.GasInfo) // Prints estimated gas used.

return nil
}

```

Using gRPC

It is not possible to generate or sign a transaction using gRPC, only to broadcast one. In order to broadcast a transaction using gRPC, you will need to generate, sign, and encode the transaction using either the CLI or programmatically with Go.

Broadcasting a Transaction

Broadcasting a transaction using the gRPC endpoint can be done by sending a `BroadcastTx` request as follows, where the `txBytes` are the protobuf-encoded bytes of a signed transaction:

```

grpcurl -plaintext \
-d '{"tx_bytes":"'{{txBytes}}','mode":"BROADCAST_MODE_SYNC"}' \
localhost:9090 \
cosmos.tx.v1beta1.Service/BroadcastTx

```

Using REST

It is not possible to generate or sign a transaction using REST, only to broadcast one. In order to broadcast a transaction using REST, you will need to generate, sign, and encode the transaction using either the CLI or programmatically with Go.

Broadcasting a Transaction

Broadcasting a transaction using the REST endpoint (served by `gRPC-gateway`) can be done by sending a POST request as follows, where the `txBytes` are the protobuf-encoded bytes of a signed transaction:

```

curl -X POST \
-H "Content-Type: application/json" \

```

```
-d'{"tx_bytes":"{{txBytes}}","mode":"BROADCAST_MODE_SYNC"}' \
localhost:1317/cosmos/tx/v1beta1/txs
```

Using CosmJS (JavaScript & TypeScript)

CosmJS aims to build client libraries in JavaScript that can be embedded in web applications. Please see

<https://cosmos.github.io/cosmjs> for more information. As of January 2021, CosmJS documentation is still work in progress.

sidebar_position: 1

Running a Testnet

:::note Synopsis The `simd testnet` subcommand makes it easy to initialize and start a simulated test network for testing purposes. :::

In addition to the commands for [running a node](#), the `simd` binary also includes a `testnet` command that allows you to start a simulated test network in-process or to initialize files for a simulated test network that runs in a separate process.

Initialize Files

First, let's take a look at the `init-files` subcommand.

This is similar to the `init` command when initializing a single node, but in this case we are initializing multiple nodes, generating the genesis transactions for each node, and then collecting those transactions.

The `init-files` subcommand initializes the necessary files to run a test network in a separate process (i.e. using a Docker container). Running this command is not a prerequisite for the `start` subcommand ([see below](#)).

In order to initialize the files for a test network, run the following command:

```
simd testnet init-files
```

You should see the following output in your terminal:

```
Successfully initialized 4 node directories
```

The default output directory is a relative `.testnets` directory. Let's take a look at the files created within the `.testnets` directory.

gentxs

The `gentxs` directory includes a genesis transaction for each validator node. Each file includes a JSON encoded genesis transaction used to register a validator node at the time of genesis. The genesis

transactions are added to the `genesis.json` file within each node directory during the initialization process.

nodes

A node directory is created for each validator node. Within each node directory is a `simd` directory. The `simd` directory is the home directory for each node, which includes the configuration and data files for that node (i.e. the same files included in the default `~/.simapp` directory when running a single node).

Start Testnet

Now, let's take a look at the `start` subcommand.

The `start` subcommand both initializes and starts an in-process test network. This is the fastest way to spin up a local test network for testing purposes.

You can start the local test network by running the following command:

```
simd testnet start
```

You should see something similar to the following:

```
acquiring test network lock
preparing test network with chain-id "chain-mtoD9v"

+++++
++      THIS MNEMONIC IS FOR TESTING PURPOSES ONLY      ++
++      DO NOT USE IN PRODUCTION          ++
++      +
++  sustain know debris minute gate hybrid stereo custom ++
++  divorce cross spoon machine latin vibrant term oblige ++
++  moment beauty laundry repeat grab game bronze truly ++
+++++

starting test network...
started test network
press the Enter Key to terminate
```

The first validator node is now running in-process, which means the test network will terminate once you either close the terminal window or you press the Enter key. In the output, the mnemonic phrase for the first validator node is provided for testing purposes. The validator node is using the same default addresses being used when initializing and starting a single node (no need to provide a `--node` flag).

Check the status of the first validator node:

```
simd status
```

Import the key from the provided mnemonic:

```
simd keys add test --recover --keyring-backend test
```

Check the balance of the account address:

```
simd q bank balances [address]
```

Use this test account to manually test against the test network.

Testnet Options

You can customize the configuration of the test network with flags. In order to see all flag options, append the `--help` flag to each command.

sidebar_position: 1

Running in Production

:::note Synopsis This section describes how to securely run a node in a public setting and/or on a mainnet on one of the many Cosmos SDK public blockchains. :::

When operating a node, full node or validator, in production it is important to set your server up securely.

:::note There are many different ways to secure a server and your node, the described steps here is one way. To see another way of setting up a server see the [run in production tutorial](#). :::

:::note This walkthrough assumes the underlying operating system is Ubuntu. :::

Server Setup

User

When creating a server most times it is created as user `root`. This user has heightened privileges on the server. When operating a node, it is recommended to not run your node as the root user.

1. Create a new user

```
sudo adduser change_me
```

2. We want to allow this user to perform sudo tasks

```
sudo usermod -aG sudo change_me
```

Now when logging into the server, the non `root` user can be used.

Go

1. Install the [Go](#) version preconized by the application.

:::warning In the past, validators [have had issues](#) when using different versions of Go. It is recommended that the whole validator set uses the version of Go that is preconized by the application. :::

Firewall

Nodes should not have all ports open to the public, this is a simple way to get DDOS'd. Secondly it is recommended by [CometBFT](#) to never expose ports that are not required to operate a node.

When setting up a firewall there are a few ports that can be open when operating a Cosmos SDK node. There is the CometBFT json-RPC, prometheus, p2p, remote signer and Cosmos SDK GRPC and REST. If the node is being operated as a node that does not offer endpoints to be used for submission or querying then a max of three endpoints are needed.

Most, if not all servers come equipped with [ufw](#). Ufw will be used in this tutorial.

1. Reset UFW to disallow all incoming connections and allow outgoing

```
sudo ufw default deny incoming  
sudo ufw default allow outgoing
```

2. Lets make sure that port 22 (ssh) stays open.

```
sudo ufw allow ssh
```

or

```
sudo ufw allow 22
```

Both of the above commands are the same.

3. Allow Port 26656 (cometbft p2p port). If the node has a modified p2p port then that port must be used here.

```
sudo ufw allow 26656/tcp
```

4. Allow port 26660 (cometbft [prometheus](#)). This acts as the applications monitoring port as well.

```
sudo ufw allow 26660/tcp
```

5. IF the node which is being setup would like to expose CometBFTs jsonRPC and Cosmos SDK GRPC and REST then follow this step. (Optional)

CometBFT JsonRPC

```
sudo ufw allow 26657/tcp
```

Cosmos SDK GRPC

```
sudo ufw allow 9090/tcp
```

Cosmos SDK REST

```
sudo ufw allow 1317/tcp
```

6. Lastly, enable ufw

```
sudo ufw enable
```

Signing

If the node that is being started is a validator there are multiple ways a validator could sign blocks.

File

File based signing is the simplest and default approach. This approach works by storing the consensus key, generated on initialization, to sign blocks. This approach is only as safe as your server setup as if the server is compromised so is your key. This key is located in the `config/priv_val_key.json` directory generated on initialization.

A second file exists that user must be aware of, the file is located in the data directory

`data/priv_val_state.json`. This file protects your node from double signing. It keeps track of the consensus keys last sign height, round and latest signature. If the node crashes and needs to be recovered this file must be kept in order to ensure that the consensus key will not be used for signing a block that was previously signed.

Remote Signer

A remote signer is a secondary server that is separate from the running node that signs blocks with the consensus key. This means that the consensus key does not live on the node itself. This increases security because your full node which is connected to the remote signer can be swapped without missing blocks.

The two most used remote signers are [tmkms](#) from [Iglusion](#) and [horcrux](#) from [Strangelove](#).

TMKMS

DEPENDENCIES

1. Update server dependencies and install extras needed.

```
sudo apt update -y && sudo apt install build-essential curl jq -y
```

2. Install Rust:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

3. Install Libusb:

```
sudo apt install libusb-1.0-0-dev
```

SETUP

There are two ways to install tmkms, from source or `cargo install`. In the examples we will cover downloading or building from source and using softsign. Softsign stands for software signing, but you could use a [yubihs](#)m as your signing key if you wish.

1. Build:

From source:

```
cd $HOME
git clone https://github.com/iqlusioninc/tmkms.git
cd $HOME/tmkms
cargo install tmkms --features=softsign
tmkms init config
tmkms softsign keygen ./config/secrets/secret_connection_key
```

or

Cargo install:

```
cargo install tmkms --features=softsign
tmkms init config
tmkms softsign keygen ./config/secrets/secret_connection_key
```

:::note To use tmkms with a yubikey install the binary with `--features=yubihs`m . :::

2. Migrate the validator key from the full node to the new tmkms instance.

```
scp user@123.456.32.123:~/simd/config/priv_validator_key.json
~/tmkms/config/secrets
```

3. Import the validator key into tmkms.

```
tmkms softsign import $HOME/tmkms/config/secrets/priv_validator_key.json
$HOME/tmkms/config/secrets/priv_validator_key
```

At this point, it is necessary to delete the `priv_validator_key.json` from the validator node and the tmkms node. Since the key has been imported into tmkms (above) it is no longer necessary on the nodes. The key can be safely stored offline.

4. Modifiy the `tmkms.toml`.

```
vim $HOME/tmkms/config/tmkms.toml
```

This example shows a configuration that could be used for soft signing. The example has an IP of 123.456.12.345 with a port of 26659 a chain_id of test-chain-waSDSe . These are items that most be modified for the usecase of tmkms and the network.

```
# CometBFT KMS configuration file
```

```

## Chain Configuration

[[chain]]
id = "osmosis-1"
key_format = { type = "bech32", account_key_prefix = "cosmospub",
consensus_key_prefix = "cosmosvalconspub" }
state_file = "/root/tmkms/config/state/priv_validator_state.json"

## Signing Provider Configuration

### Software-based Signer Configuration

[[providers.softsign]]
chain_ids = ["test-chain-waSDSe"]
key_type = "consensus"
path = "/root/tmkms/config/secrets/priv_validator_key"

## Validator Configuration

[[validator]]
chain_id = "test-chain-waSDSe"
addr = "tcp://123.456.12.345:26659"
secret_key = "/root/tmkms/config/secrets/secret_connection_key"
protocol_version = "v0.34"
reconnect = true

```

5. Set the address of the tmkms instance.

```

vim $HOME/.simd/config/config.toml

priv_validator_laddr = "tcp://0.0.0.0:26659"

```

:::tip The above address is set to `0.0.0.0` but it is recommended to set the tmkms server to secure the startup :::

:::tip It is recommended to comment or delete the lines that specify the path of the validator key and validator:

```

# Path to the JSON file containing the private key to use as a validator in the
# consensus protocol
# priv_validator_key_file = "config/priv_validator_key.json"

# Path to the JSON file containing the last sign state of a validator
# priv_validator_state_file = "data/priv_validator_state.json"

```

:::

6. Start the two processes.

```

tmkms start -c $HOME/tmkms/config/tmkms.toml

```

```
simd start
```

sidebar_position: 1

Protocol Buffers

It is known that Cosmos SDK uses protocol buffers extensively, this document is meant to provide a guide on how it is used in the cosmos-sdk.

To generate the proto file, the Cosmos SDK uses a docker image, this image is provided to all to use as well. The latest version is `ghcr.io/cosmos/proto-builder:0.12.x`

Below is the example of the Cosmos SDK's commands for generating, linting, and formatting protobuf files that can be reused in any applications makefile.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/Makefile#L411-L432
```

The script used to generate the protobuf files can be found in the `scripts/` directory.

```
https://github.com/cosmos/cosmos-sdk/blob/v0.50.0-alpha.0/scripts/protocgen.sh
```

Buf

[Buf](#) is a protobuf tool that abstracts the needs to use the complicated `protoc` toolchain on top of various other things that ensure you are using protobuf in accordance with the majority of the ecosystem. Within the cosmos-sdk repository there are a few files that have a buf prefix. Lets start with the top level and then dive into the various directories.

Workspace

At the root level directory a workspace is defined using [buf workspaces](#). This helps if there are one or more protobuf containing directories in your project.

Cosmos SDK example:

```
https://github.com/cosmos/cosmos-sdk/blob/main/buf.work.yaml#L6-L9
```

Proto Directory

Next is the `proto/` directory where all of our protobuf files live. In here there are many different buf files defined each serving a different purpose.

```
|__ README.md  
|__ buf.gen.gogo.yaml  
|__ buf.gen.pulsar.yaml  
|__ buf.gen.swagger.yaml  
|__ buf.lock
```

```
└── buf.md
└── buf.yaml
└── cosmos
└── tendermint
```

The above diagram all the files and directories within the Cosmos SDK `proto/` directory.

buf.gen.gogo.yaml

`buf.gen.gogo.yaml` defines how the protobuf files should be generated for use with in the module. This file uses [gogoproto](#), a separate generator from the google go-proto generator that makes working with various objects more ergonomic, and it has more performant encode and decode steps

```
https://github.com/cosmos/cosmos-sdk/blob/main/proto/buf.gen.gogo.yaml#L1-L9
```

:::tip Example of how to define `.gen` files can be found [here](#) :::

buf.gen.pulsar.yaml

`buf.gen.pulsar.yaml` defines how protobuf files should be generated using the [new golang apiv2 of protobuf](#). This generator is used instead of the google go-proto generator because it has some extra helpers for Cosmos SDK applications and will have more performant encode and decode than the google go-proto generator. You can follow the development of this generator [here](#).

```
https://github.com/cosmos/cosmos-sdk/blob/main/proto/buf.gen.pulsar.yaml#L1-L18
```

:::tip Example of how to define `.gen` files can be found [here](#) :::

buf.gen.swagger.yaml

`buf.gen.swagger.yaml` generates the swagger documentation for the query and messages of the chain. This will only define the REST API end points that were defined in the query and msg servers. You can find examples of this [here](#)

```
https://github.com/cosmos/cosmos-sdk/blob/main/proto/buf.gen.swagger.yaml#L1-L6
```

:::tip Example of how to define `.gen` files can be found [here](#) :::

buf.lock

This is a autogenerated file based off the dependencies required by the `.gen` files. There is no need to copy the current one. If you depend on cosmos-sdk proto definitions a new entry for the Cosmos SDK will need to be provided. The dependency you will need to use is `buf.build/cosmos/cosmos-sdk`.

```
https://github.com/cosmos/cosmos-sdk/blob/main/proto/buf.lock#L1-L16
```

buf.yaml

`buf.yaml` defines the [name of your package](#), which [breakage checker](#) to use and how to [lint your protobuf files](#).

```
https://github.com/cosmos/cosmos-sdk/blob/main/proto/buf.yaml#L1-L24
```

We use a variety of linters for the Cosmos SDK protobuf files. The repo also checks this in ci.

A reference to the github actions can be found [here](#)

```
https://github.com/cosmos/cosmos-sdk/blob/main/.github/workflows/proto.yml#L1-L32
```

sidebar_position: 0

Tools

This section provides documentation on various tooling maintained by the SDK team. This includes tools for development, operating a node, and ease of use of a Cosmos SDK chain.

CLI Tools

- [Cosmovisor](#)
- [Confix](#)
- [Hubl](#)
- [Rosetta](#)

Other Tools

- [Protocol Buffers](#)

RFC Creation Process

1. Copy the `rfc-template.md` file. Use the following filename pattern: `rfc-next_number-title.md`
2. Create a draft Pull Request if you want to get an early feedback.
3. Make sure the context and a solution is clear and well documented.
4. Add an entry to a list in the [README](#) file.
5. Create a Pull Request to propose a new ADR.

What is an RFC?

An RFC is a sort of async whiteboarding session. It is meant to replace the need for a distributed team to come together to make a decision. Currently, the Cosmos SDK team and contributors are distributed around the world. The team conducts working groups to have a synchronous discussion and an RFC can be used to capture the discussion for a wider audience to better understand the changes that are coming to the software.

The main difference the Cosmos SDK is defining as a differentiation between RFC and ADRs is that one is to come to consensus and circulate information about a potential change or feature. An ADR is used if there is already consensus on a feature or change and there is not a need to articulate the change coming to the software. An ADR will articulate the changes and have a lower amount of communication .

RFC life cycle

RFC creation is an **iterative** process. An RFC is meant as a distributed collaboration session, it may have many comments and is usually the bi-product of no working group or synchronous communication

1. Proposals could start with a new GitHub Issue, be a result of existing Issues or a discussion.
2. An RFC doesn't have to arrive to `main` with an *accepted* status in a single PR. If the motivation is clear and the solution is sound, we **SHOULD** be able to merge it and keep a *proposed* status. It's preferable to have an iterative approach rather than long, not merged Pull Requests.
3. If a *proposed* RFC is merged, then it should clearly document outstanding issues either in the RFC document notes or in a GitHub Issue.
4. The PR **SHOULD** always be merged. In the case of a faulty RFC, we still prefer to merge it with a *rejected* status. The only time the RFC **SHOULD NOT** be merged is if the author abandons it.
5. Merged RFCs **SHOULD NOT** be pruned.
6. If there is consensus and enough feedback then the RFC can be accepted.

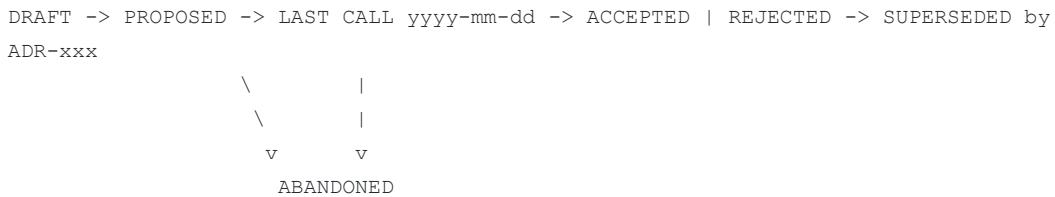
Note: An RFC is written when there is no working group or team session on the problem. RFC's are meant as a distributed white boarding session. If there is a working group on the proposal there is no need to have an RFC as there is synchronous whiteboarding going on.

RFC status

Status has two components:

{CONSENSUS STATUS}

Consensus Status



- `DRAFT` : [optional] an ADR which is work in progress, not being ready for a general review. This is to present an early work and get an early feedback in a Draft Pull Request form.
- `PROPOSED` : an ADR covering a full solution architecture and still in the review - project stakeholders haven't reached an agreed yet.
- `LAST CALL <date for the last call>` : [optional] clear notify that we are close to accept updates. Changing a status to `LAST CALL` means that social consensus (of Cosmos SDK maintainers) has been reached and we still want to give it a time to let the community react or analyze.
- `ACCEPTED` : ADR which will represent a currently implemented or to be implemented architecture design.

- **REJECTED** : ADR can go from PROPOSED or ACCEPTED to rejected if the consensus among project stakeholders will decide so.
- **SUPERSEDED by ADR-xxx** : ADR which has been superseded by a new ADR.
- **ABANDONED** : the ADR is no longer pursued by the original authors.

Language used in RFC

- The background/goal should be written in the present tense.
 - Avoid using a first, personal form.
-

sidebar_position: 1

Requests for Comments

A Request for Comments (RFC) is a record of discussion on an open-ended topic related to the design and implementation of the Cosmos SDK, for which no immediate decision is required.

The purpose of an RFC is to serve as a historical record of a high-level discussion that might otherwise only be recorded in an ad-hoc way (for example, via gists or Google docs) that are difficult to discover for someone after the fact. An RFC *may* give rise to more specific architectural *decisions* for the Cosmos SDK, but those decisions must be recorded separately in [Architecture Decision Records \(ADR\)](#).

As a rule of thumb, if you can articulate a specific question that needs to be answered, write an ADR. If you need to explore the topic and get input from others to know what questions need to be answered, an RFC may be appropriate.

RFC Content

An RFC should provide:

- A **changelog**, documenting when and how the RFC has changed.
- An **abstract**, briefly summarizing the topic so the reader can quickly tell whether it is relevant to their interest.
- Any **background** a reader will need to understand and participate in the substance of the discussion (links to other documents are fine here).
- The **discussion**, the primary content of the document.

The [rfc-template.md](#) file includes placeholders for these sections.

Table of Contents

- [RFC-002: Zero Copy Encoding](#)

RFC 001: Transaction Validation

Changelog

- 2023-03-12: Proposed

Background

Transation Validation is crucial to a functioning state machine. Within the Cosmos SDK there are two validation flows, one is outside the message server and the other within. The flow outside of the message server is the `ValidateBasic` function. It is called in the antehandler on both `CheckTx` and `DeliverTx`. There is an overhead and sometimes duplication of validation within these two flows. This extra validation provides an additional check before entering the mempool.

With the deprecation of `GetSigners` we have the optionality to remove `sdk.Msg` and the `ValidateBasic` function.

With the separation of CometBFT and Cosmos-SDK, there is a lack of control of what transactions get broadcasted and included in a block. This extra validation in the antehandler is meant to help in this case. In most cases the transaction is or should be simulated against a node for validation. With this flow transactions will be treated the same.

Proposal

The acceptance of this RFC would move validation within `ValidateBasic` to the message server in modules, update tutorials and docs to remove mention of using `ValidateBasic` in favour of handling all validation for a message where it is executed.

We can and will still support the `Validatebasic` function for users and provide an extension interface of the function once `sdk.Msg` is deprecated.

Note: This is how messages are handled in VMs like Ethereum and CosmWasm.

Consequences

The consequence of updating the transaction flow is that transaction that may have failed before with the `ValidateBasic` flow will now be included in a block and fees charged.

RFC 002: Zero-Copy Encoding

Changelog

- 2022-03-08: Initial draft

Background

When the SDK originally migrated to [protobuf state encoding](#), zero-copy encodings such as [Cap'n Proto](#) and [FlatBuffers](#) were considered. We considered how a zero-copy encoding could be beneficial for interoperability with modules and scripts in other languages and VMs. However, protobuf was still chosen because the maturity of its ecosystem and tooling was much higher and the client experience and performance were considered the highest priorities.

In [ADR 033: Protobuf-based Inter-Module Communication](#), the idea of cross-language/VM inter-module communication was considered again. And in the discussions surrounding [ADR 054: Semver Compatible SDK Modules](#), it was determined that multi-language/VM support in the SDK is a near term priority.

While we could do cross-language/VM inter-module communication with protobuf binary or even JSON, the performance overhead is deemed to be too high because:

- we are proposing replacing keeper calls with inter-module message calls and the overhead of even the inter-module routing checks has come into question by some SDK users without even

considering the possible overhead of encoding. Effectively we would be replacing function calls with encoding. One of the SDK's primary objectives currently is improving performance, and we want to avoid inter-module calls from becoming a big step backward.

- we want Rust code to be able to operate in highly resource constrained virtual machines so whatever we can do to reduce performance overhead as well as the size of generated code will make it easier and more feasible to deploy first-class integrations with these virtual machines.

Thus, the agreement when the [ADR 054](#) working group concluded was to pursue a performant zero-copy encoding which is suitable for usage in highly resource constrained environments.

Proposal

This RFC proposes a zero-copy encoding that is derived from the schema definitions defined in .proto files in the SDK and all app chains. This would result in a new code generator for that supports both this zero-copy encoding as well as the existing protobuf binary and JSON encodings as well as the [google.golang.org/protobuf API](https://google.github.io/protobuf/). To make this zero-copy encoding work, a number of changes are needed to how we manage the versioning of protobuf messages that should address other concerns raised in [ADR 054](#). The API for using protobuf in golang would also change and this will be described in the [code generation](#) section along with a proposed Rust code generator.

An alternative approach to building a zero-copy encoding based on protobuf schemas would be to switch to FlatBuffers or Cap'n Proto directly. However, this would require a complete rewrite of the SDK and all app chains. Places this burden on the ecosystem would not be a wise choice when creating a zero-copy encoding compatible with all our existing types and schemas is feasible. In the future, we may consider a native schema language for this encoding that is more natural and succinct for its rules, but for now we are assuming that it is best to continue supporting the existing protobuf based workflow.

Also, we are not proposing a new encoding for transactions or gRPC query servers. From a client API perspective nothing would change. The SDK would be capable of marshaling any message to and from protobuf binary and this zero-copy encoding as needed.

Furthermore, migrating to the **new golang generated code would be 100% opt-in** because the inter-module router will simply marshal existing gogo proto generated types to/from the zero-copy encoding when needed. So migrating to the new code generator would provide a performance benefit, but would not be required.

In addition to supporting first-class Cosmos SDK modules defined in other languages and VMs, this encoding is intended to be useful for user-defined code executing in a VM. To satisfy this, this encoding is designed to enable proper bounds checking on all memory access at the expense of introducing some error return values in generated code.

New Protobuf Linting and Breaking Change Rules

This zero-copy encoding places some additional requirements on the definition and maintenance of protobuf schemas.

No New Fields Can Be Added To Existing Messages

The biggest change is that it will be invalid to add a new field to an existing message and a breaking change detector will need to be created which augments [buf breaking](#) to detect this.

The reasons for this are two-fold:

- from an API compatibility perspective, adding a new field to an existing message is actually a state machine breaking change which in [ADR 020](#) required us to add an unknown field detector. Furthermore, in [ADR 054](#) this "feature" of protobuf poses one of the biggest problems for correct forward compatibility between different versions of the same module.
- not allowing new fields in existing messages makes the generated code in languages like Rust (which is currently our highest priority target), much simpler and more performant because we can assume a fixed size struct gets allocated. If new fields can be added to existing messages, we need to encode the number of fields into the message and then do runtime checks. So this both increases memory layers and requires another layout of indirection. With the encoding proposed below, "plain old Rust structs" (used with some special field types) can be used.

Instead of adding new fields to existing messages, APIs can add new messages to existing packages or create new packages with new versions of the messages. Also, we are not restricting the addition of cases to `oneof`s or values to `enum`s. All of these cases are easier to detect at runtime with standard `switch` statements than the addition of new fields.

Additional Linting Rules

The following additional rules will be enforced by a linter that complements [buf lint](#):

- all message fields must be specified in continuous ascending order starting from `1`
- all enums must be specified in continuous ascending order starting from `0` - otherwise it is too complex to check at runtime whether an enum value is unknown. An alternative would be to make adding new values to existing enums breaking
- all enum values must be `<= 255`. Any enum in a blockchain application which needs more than 256 values is probably doing something very wrong.
- all `oneof`'s must be the *only* element in their containing message and must start at field number `1` and be added in continuous ascending order - this makes it possible to quickly check for unknown values
- all `oneof` field numbers must be `<= 255`. Any `oneof` which needs more field cases is probably doing something very wrong.

These requirements make the encoding and generated code simpler.

Encoding

Buffers and Memory Management

By default, this encoding attempts to use a single fixed size encoding buffer of 64kb. This imposes a limit on the maximum size of a message that can be encoded. In the context of a message passing protocol for blockchains, this is generally a reasonable limit and the only known valid use case for exceeding it is to store user-uploaded byte code for execution in VMs. To accommodate this, large `string` and `bytes` values can be encoded in additional standalone buffers if needed. Still, the body of a message included all scalar and message fields must fit inside the 64kb buffer.

While this design decision greatly simplifies the encoding and decoding logic, as well as the complexity of generated code, it does mean that APIs will need to do proper bounds checking when writing data that is not fixed size and return errors.

The term `Root` is used to refer to the main 64kb buffer plus any additional large `string / bytes` buffers that are allocated.

Scalar Encoding

- `bool`s are encoded as 1 byte - `0` or `1`

- `uint32`, `int32`, `sint32`, `fixed32`, `sfixed32` are encoded as 4 bytes by default
- `uint64`, `int64`, `sint64`, `fixed64`, `sfixed64` are encoded as 8 bytes by default
- `enum` s are encoded as 1 byte and values *MUST* be in the range of `0` to `255`.
- all scalars declared as `optional` are prefixed with 1 additional byte whose value is `0` or `1` to indicate presence

All multibyte integers are encoded as little-endian which is by far the most common native byte order for modern CPUs. Signed integers always use two's complement encoding.

Message Encoding

By default, messages field are encoded inline as structs. Meaning that if a message struct takes 8 bytes then its inline field in another struct will add 8 bytes to that struct size.

`optional` message fields will be prefixed by 1 byte to indicate presence. (Alternatively, we could encode optional message fields as pointers (see below) if the desire is to save memory when they are rarely used needed.)

Oneof's

`oneof` s are encoded as a combination of a `uint8` discriminant field and memory that is as large as the largest member field. `oneof` field numbers *MUST* be between `1` and `255`.

```
message Foo {
  oneof sum {
    bool x = 1;
    int32 y = 2;
  }
}
```

A discriminant of `0` indicates that the field is not set.

Pointer Types: Bytes and Strings and Repeated fields

A pointer is an 16-bit unsigned integer that points to an offset in the current memory buffer or to another memory buffer. If the bit mask `0xFF00` on the is unset, then the pointer points to an offset in the main 64kb memory buffer. If that bit mask is set, then the pointer points to a large `string` or `bytes` buffer. Up to 256 such buffers can be referenced in a single `Root`. The pointer `0` indicates that a field is not defined.

`bytes`, `string` s and repeated fields are encoded as pointers to a memory location that is prefixed with the length of the `bytes`, `string` or repeated field value. If the referenced memory location is in the main 64kb memory buffer, then this length prefix will be a 16-bit unsigned integer. If the referenced memory location is a large `string` or `bytes` buffer, then this length prefix will be a 32-bit unsigned integer.

Any s

`Any` s are encoded as a pointer to the type URL string and a pointer to the start of the message specified by the type URL.

Maps

Maps are not supported.

Extended Encoding Options

We may choose to allow customizing the encoding of fields so that they take up less space.

For example, we could allow 8-bit or 16-bit integers: `int32 x = 1 [(cosmos_proto.int16) = true]` would indicate that the field only needs 2 bytes

Or we could allow `string`, `bytes` or `repeated` fields to have a fixed size rather than being encoded as pointers to a variable-length value: `string y = 2 [(cosmos_proto.fixed_size) = 3]` could indicate that this is a fixed width 3 byte string

If we choose to enable these encoding options, changing these options would be a breaking change that needs to be prevented by the breaking change detector.

Generated Code

We will describe the generated Go and Rust code using this example protobuf file:

```
message Foo {
    int32 x = 1;
    optional uint32 y = 2;
    string z = 3;
    Bar bar = 4;
    repeated Bar bars = 5;
}

message Bar {
    ABC abc = 1;
    Baz baz = 2;
    repeated uint32 xs = 3;
}

message Baz {
    oneof sum {
        uint32 x = 1;
        string y = 2;
    }
}

enum ABC {
    A = 0;
    B = 1;
    C = 2;
    D = 3;
}
```

Go

In golang, the generated code would not expose any exported struct fields, but rather getters and setters as an interface or struct methods, ex:

```

type Foo interface {
    X() int32
    SetX(int32)
    Y() zpb.Option[uint32]
    SetY(zpb.Option[uint32])
    Z() (string, error)
    SetZ(string) error
    Bar() Bar
    Bars() (zpb.Array[Bar], error)
}

type Bar interface {
    Abc() ABC
    SetAbc(ABC) Bar
    Baz() Baz
    Xs() (zpb.ScalarArray[uint32], error)
}

type Baz interface {
    Case() Baz_case
    GetX() uint32
    SetX(uint32)
    GetY() (string, error)
    SetY(string)
}

type Baz_case int32
const (
    Baz_X Baz_case = 0
    Baz_Y Baz_case = 1
)

type ABC int32
const (
    ABC_A ABC = 0
    ABC_B ABC = 1
    ABC_C ABC = 2
    ABC_D ABC = 3
)

```

Special types `zpb.Option`, `zpb.Array` and `zpb.ScalarArray` are used to represent optional and repeated fields respectively. These types would be included in the runtime library (called `zpb` here for zero-copy protobuf) and would have an API like this:

```

type Option[T] interface {
    IsSet() bool
    Value() T
}

type Array[T] interface {

```

```

    InitWithLength(int) error
    Len() int
    Get(int) T
}

type ScalarArray[T] interface {
    Array[T]
    Set(int, T)
}

```

Arrays in particular would not be resizable, but would be initialized with a fixed length. This is to ensure that arrays can be written to the underlying buffer in a linear way.

In golang, buffers would be managed transparently under the hood by the first message initialized, and usage of this generated code might look like this:

```

foo := NewFoo()
foo.SetX(1)
foo.SetY(zpb.NewOption[uint32](2))
err := foo.SetZ("hello")
if err != nil {
    panic(err)
}

bar := foo.Bar()
bar.Baz().SetX(3)

xs, err = bar.Xs()
if err != nil {
    panic(err)
}
xs.InitWithLength(2)
xs.Set(0, 0)
xs.Set(1, 2)

bars, err = foo.Bars()
if err != nil {
    panic(err)
}
bars.InitWithLength(3)
bars.Get(0).Baz().SetY("hello")
bars.Get(1).SetAbc(ABC_B)
bars.Get(2).Baz().SetX(4)

```

Under the hood the generated code would manage memory buffers on its own. The usage of `oneof`s is a bit easier than the existing go generated code (as with `bar.Baz()` above). And rather than using setters on embedded messages, we simply get the field (already allocated) and set its fields (as in the case of `foo.Bar()` above or the repeated field `foo.Bars()`). Whenever a field is stored with a pointer (`string`, `bytes`, and `repeated` fields), there is always an error returned on the getter to do proper bounds checking on the buffer.

Rust

This encoding should allow generating native structs in Rust that are annotated with `#[repr(C, align(1))]`. It should be fairly natural to use from Rust with a key difference that memory buffers (called `Root`s) must be manually allocated and passed into any pointer type.

Here is some example code that uses library types `Option`, `Enum`, `String`, `OneOf` and `Repeated` as well as little-endian integer types from [rend](#):

```
#[repr(C, align(1))]
struct Foo {
    x: rend::i32_le,
    y: cosmos_proto::Option<rend::u32_le>,
    z: cosmos_proto::String, // String wraps a pointer to a string
    bar: Bar
}

#[repr(C, align(1))]
struct Bar {
    abc: cosmos_proto::Enum<ABC, 3>, // the Enum wrapper allows us to distinguish
    undefined and defined values of ABC at runtime. 3 is specified as the max value of
    ABC.
    baz: cosmos_proto::OneOf<Baz, 2>, // the OneOf wrapper allows distinguished
    undefined values of Baz at runtime. 2 is specified as the max field value of Baz.
    xs: cosmos_proto::Repeated<rend::u32_le> // Repeated wraps a pointer to repeated
    fields
}

#[repr(u8)]
enum ABC {
    A = 0,
    B = 1,
    C = 2,
    D = 3,
}

#[repr(C, u8)]
enum Baz {
    Empty, // all oneof's have a case for Empty if they are unset
    X(rend::u32_le),
    Y(cosmos_proto::String)
}
```

Example usage (which does the exact same thing as the go example above) would be:

```
let mut root = Root<Foo>::new();
let mut foo = root.get_mut();
foo.x = 1.into();
foo.y = Some(2.into());
foo.z.set(root.new_string("hello")?); // could return an allocation error
```

```
foo.bar.baz = Baz::X(3.into());  
  
foo.bar.xs.init_with_size(&mut root, 2)?; // could return an allocation error  
foo.bar.xs[0] = 0.into();  
foo.bar.xs[1] = 2.into();  
  
foo.bars.init_with_size(&mut root, 3)?; // could return an allocation error  
foo.bars[0].baz = Baz::Y(root.new_string("hello")?); // could return an allocation  
error  
foo.bars[1].abc = ABC::B;  
foo.bars[2].baz = Baz::X(4.into());
```

Abandoned Ideas (Optional)

References

Discussion

RFC {RFC-NUMBER}: {TITLE}

Changelog

- {date}: {changelog}

Background

The next section is the "Background" section. This section should be at least two paragraphs and can take up to a whole page in some cases. The guiding goal of the background section is: as a newcomer to this project (new employee, team transfer), can I read the background section and follow any links to get the full context of why this change is necessary?

If you can't show a random engineer the background section and have them acquire nearly full context on the necessity for the RFC, then the background section is not full enough. To help achieve this, link to prior RFCs, discussions, and more here as necessary to provide context so you don't have to simply repeat yourself.

Proposal

The next required section is "Proposal" or "Goal". Given the background above, this section proposes a solution. This should be an overview of the "how" for the solution, but for details further sections will be used.

Abandoned Ideas (Optional)

As RFCs evolve, it is common that there are ideas that are abandoned. Rather than simply deleting them from the document, you should try to organize them into sections that make it clear they're abandoned while explaining why they were abandoned.

When sharing your RFC with others or having someone look back on your RFC in the future, it is common to walk the same path and fall into the same pitfalls that we've since matured from. Abandoned ideas are a way to recognize that path and explain the pitfalls and why they were abandoned.

Decision

This section describes alternative designs to the chosen design. This section is important and if an ADR does not have any alternatives then it should be considered that the ADR was not thought through.

Consequences (optional)

This section describes the resulting context, after applying the decision. All consequences should be listed here, not just the "positive" ones. A particular decision may have positive, negative, and neutral consequences, but all of them affect the team and project in the future.

Backwards Compatibility

All ADRs that introduce backwards incompatibilities must include a section describing these incompatibilities and their severity. The ADR must explain how the author proposes to deal with these incompatibilities. ADR submissions without a sufficient backwards compatibility treatise may be rejected outright.

Positive

{positive consequences}

Negative

{negative consequences}

Neutral

{neutral consequences}

References

Links to external materials needed to follow the discussion may be added here.

In addition, if the discussion in a request for comments leads to any design decisions, it may be helpful to add links to the ADR documents here after the discussion has settled.

Discussion

This section contains the core of the discussion.

There is no fixed format for this section, but ideally changes to this section should be updated before merging to reflect any discussion that took place on the PR that made those changes.

sidebar_position: 1

Specifications

This directory contains specifications for the modules of the Cosmos SDK as well as Interchain Standards (ICS) and other specifications.

Cosmos SDK applications hold this state in a Merkle store. Updates to the store may be made during transactions and at the beginning and end of every block.

Cosmos SDK specifications

- [Store](#) - The core Merkle store that holds the state.
- [Bech32](#) - Address format for Cosmos SDK applications.

Modules specifications

Go the [module directory](#).

CometBFT

For details on the underlying blockchain and p2p protocols, see the [CometBFT specification](#).

Specification of Modules

This file intends to outline the common structure for specifications within this directory.

Tense

For consistency, specs should be written in passive present tense.

Pseudo-Code

Generally, pseudo-code should be minimized throughout the spec. Often, simple bulleted-lists which describe a function's operations are sufficient and should be considered preferable. In certain instances, due to the complex nature of the functionality being described pseudo-code may be the most suitable form of specification. In these cases use of pseudo-code is permissible, but should be presented in a concise manner, ideally restricted to only the complex element as a part of a larger description.

Common Layout

The following generalized `README` structure should be used to breakdown specifications for modules. The following list is nonbinding and all sections are optional.

- `# {Module Name}` - overview of the module
- `## Concepts` - describe specialized concepts and definitions used throughout the spec
- `## State` - specify and describe structures expected to marshalled into the store, and their keys
- `## State Transitions` - standard state transition operations triggered by hooks, messages, etc.
- `## Messages` - specify message structure(s) and expected state machine behaviour(s)
- `## Begin Block` - specify any begin-block operations

- `## End Block` - specify any end-block operations
- `## Hooks` - describe available hooks to be called by/from this module
- `## Events` - list and describe event tags used
- `## Client` - list and describe CLI commands and gRPC and REST endpoints
- `## Params` - list all module parameters, their types (in JSON) and examples
- `## Future Improvements` - describe future improvements of this module
- `## Tests` - acceptance tests
- `## Appendix` - supplementary details referenced elsewhere within the spec

Notation for key-value mapping

Within `## State` the following notation `->` should be used to describe key to value mapping:

```
key -> value
```

to represent byte concatenation the `|` may be used. In addition, encoding type may be specified, for example:

```
0x00 | addressBytes | address2Bytes -> amino(value_object)
```

Additionally, index mappings may be specified by mapping to the `nil` value, for example:

```
0x01 | address2Bytes | addressBytes -> nil
```

What is an SDK standard?

An SDK standard is a design document describing a particular protocol, standard, or feature expected to be used by the Cosmos SDK. A SDK standard should list the desired properties of the standard, explain the design rationale, and provide a concise but comprehensive technical specification. The primary author is responsible for pushing the proposal through the standardization process, soliciting input and support from the community, and communicating with relevant stakeholders to ensure (social) consensus.

Sections

A SDK standard consists of:

- a synopsis,
- overview and basic concepts,
- technical specification,
- history log, and
- copyright notice.

All top-level sections are required. References should be included inline as links, or tabulated at the bottom of the section if necessary. Included sub-sections should be listed in the order specified below.

Table Of Contents

Provide a table of contents at the top of the file to assist readers.

Synopsis

The document should include a brief (~200 word) synopsis providing a high-level description of and rationale for the specification.

Overview and basic concepts

This section should include a motivation sub-section and a definitions sub-section if required:

- *Motivation* - A rationale for the existence of the proposed feature, or the proposed changes to an existing feature.
- *Definitions* - A list of new terms or concepts utilized in the document or required to understand it.

System model and properties

This section should include an assumptions sub-section if any, the mandatory properties sub-section, and a dependencies sub-section. Note that the first two sub-section are tightly coupled: how to enforce a property will depend directly on the assumptions made. This sub-section is important to capture the interactions of the specified feature with the "rest-of-the-world", i.e., with other features of the ecosystem.

- *Assumptions* - A list of any assumptions made by the feature designer. It should capture which features are used by the feature under specification, and what do we expect from them.
- *Properties* - A list of the desired properties or characteristics of the feature specified, and expected effects or failures when the properties are violated. In case it is relevant, it can also include a list of properties that the feature does not guarantee.
- *Dependencies* - A list of the features that use the feature under specification and how.

Technical specification

This is the main section of the document, and should contain protocol documentation, design rationale, required references, and technical details where appropriate. The section may have any or all of the following sub-sections, as appropriate to the particular specification. The API sub-section is especially encouraged when appropriate.

- *API* - A detailed description of the feature's API.
- *Technical Details* - All technical details including syntax, diagrams, semantics, protocols, data structures, algorithms, and pseudocode as appropriate. The technical specification should be detailed enough such that separate correct implementations of the specification without knowledge of each other are compatible.
- *Backwards Compatibility* - A discussion of compatibility (or lack thereof) with previous feature or protocol versions.
- *Known Issues* - A list of known issues. This sub-section is specially important for specifications of already in-use features.
- *Example Implementation* - A concrete example implementation or description of an expected implementation to serve as the primary reference for implementers.

History

A specification should include a history section, listing any inspiring documents and a plaintext log of significant changes.

See an example history section [below](#).

Copyright

A specification should include a copyright section waiving rights via [Apache 2.0](#).

Formatting

General

Specifications must be written in GitHub-flavoured Markdown.

For a GitHub-flavoured Markdown cheat sheet, see [here](#). For a local Markdown renderer, see [here](#).

Language

Specifications should be written in Simple English, avoiding obscure terminology and unnecessary jargon.

For excellent examples of Simple English, please see the [Simple English Wikipedia](#).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in specifications are to be interpreted as described in [RFC 219](#).

Pseudocode

Pseudocode in specifications should be language-agnostic and formatted in a simple imperative standard, with line numbers, variables, simple conditional blocks, for loops, and English fragments where necessary to explain further functionality such as scheduling timeouts. LaTeX images should be avoided because they are difficult to review in diff form.

Pseudocode for structs can be written in a simple language like Typescript or golang, as interfaces.

Example Golang pseudocode struct:

```
type CacheKVStore interface {
    cache: map[Key]Value
    parent: KVStore
    deleted: Key
}
```

Pseudocode for algorithms should be written in simple Golang, as functions.

Example pseudocode algorithm:

```
func get(
    store CacheKVStore,
    key Key) Value {

    value = store.cache.get(Key)
    if (value !== null) {
        return value
    } else {
        value = store.parent.get(key)
        store.cache.set(key, value)
        return value
    }
}
```

History

This specification was significantly inspired by and derived from IBC's [ICS](#), which was in turn derived from Ethereum's [EIP 1](#).

Nov 24, 2022 - Initial draft finished and submitted as a PR

Copyright

All content herein is licensed under [Apache 2.0](#).

Cosmos ICS

- [ICS030 - Signed Messages](#)

ICS 030: Cosmos Signed Messages

TODO: Replace with valid ICS number and possibly move to new location.

- [Changelog](#)
- [Abstract](#)
- [Preliminary](#)
- [Specification](#)
- [Future Adaptations](#)
- [API](#)
- [References](#)

Status

Proposed.

Changelog

Abstract

Having the ability to sign messages off-chain has proven to be a fundamental aspect of nearly any blockchain. The notion of signing messages off-chain has many added benefits such as saving on computational costs and reducing transaction throughput and overhead. Within the context of the Cosmos, some of the major applications of signing such data includes, but is not limited to, providing a cryptographic secure and verifiable means of proving validator identity and possibly associating it with some other framework or organization. In addition, having the ability to sign Cosmos messages with a Ledger or similar HSM device.

A standardized protocol for hashing, signing, and verifying messages that can be implemented by the Cosmos SDK and other third-party organizations is needed. Such a standardized protocol subscribes to the following:

- Contains a specification of human-readable and machine-verifiable typed structured data
- Contains a framework for deterministic and injective encoding of structured data
- Utilizes cryptographic secure hashing and signing algorithms
- A framework for supporting extensions and domain separation

- Is invulnerable to chosen ciphertext attacks
- Has protection against potentially signing transactions a user did not intend to

This specification is only concerned with the rationale and the standardized implementation of Cosmos signed messages. It does **not** concern itself with the concept of replay attacks as that will be left up to the higher-level application implementation. If you view signed messages in the means of authorizing some action or data, then such an application would have to either treat this as idempotent or have mechanisms in place to reject known signed messages.

Preliminary

The Cosmos message signing protocol will be parameterized with a cryptographic secure hashing algorithm `SHA-256` and a signing algorithm `S` that contains the operations `sign` and `verify` which provide a digital signature over a set of bytes and verification of a signature respectively.

Note, our goal here is not to provide context and reasoning about why necessarily these algorithms were chosen apart from the fact they are the defacto algorithms used in CometBFT and the Cosmos SDK and that they satisfy our needs for such cryptographic algorithms such as having resistance to collision and second pre-image attacks, as well as being [deterministic](#) and [uniform](#).

Specification

CometBFT has a well established protocol for signing messages using a canonical JSON representation as defined [here](#).

An example of such a canonical JSON structure is CometBFT's vote structure:

```
type CanonicalJSONVote struct {
    ChainID      string           `json:"@chain_id"`
    Type         string           `json:@type"`
    BlockID     CanonicalJSONBlockID `json:"block_id"`
    Height       int64            `json:"height"`
    Round        int              `json:"round"`
    Timestamp    string           `json:"timestamp"`
    VoteType     byte             `json:"type"`
}
```

With such canonical JSON structures, the specification requires that they include meta fields: `@chain_id` and `@type`. These meta fields are reserved and must be included. They are both of type `string`. In addition, fields must be ordered in lexicographically ascending order.

For the purposes of signing Cosmos messages, the `@chain_id` field must correspond to the Cosmos chain identifier. The user-agent should **refuse** signing if the `@chain_id` field does not match the currently active chain! The `@type` field must equal the constant `"message"`. The `@type` field corresponds to the type of structure the user will be signing in an application. For now, a user is only allowed to sign bytes of valid ASCII text ([see here](#)). However, this will change and evolve to support additional application-specific structures that are human-readable and machine-verifiable ([see Future Adaptations](#)).

Thus, we can have a canonical JSON structure for signing Cosmos messages using the [JSON schema](#) specification as such:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "$id": "cosmos/signing/typeData/schema",
  "title": "The Cosmos signed message typed data schema.",
  "type": "object",
  "properties": {
    "@chain_id": {
      "type": "string",
      "description": "The corresponding Cosmos chain identifier.",
      "minLength": 1
    },
    "@type": {
      "type": "string",
      "description": "The message type. It must be 'message'.",
      "enum": [
        "message"
      ]
    },
    "text": {
      "type": "string",
      "description": "The valid ASCII text to sign.",
      "pattern": "^\x20-\x7E]+$",
      "minLength": 1
    }
  },
  "required": [
    "@chain_id",
    "@type",
    "text"
  ]
}
```

e.g.

```
{
  "@chain_id": "1",
  "@type": "message",
  "text": "Hello, you can identify me as XYZ on keybase."
}
```

Future Adaptations

As applications can vary greatly in domain, it will be vital to support both domain separation and human-readable and machine-verifiable structures.

Domain separation will allow for application developers to prevent collisions of otherwise identical structures. It should be designed to be unique per application use and should directly be used in the signature encoding itself.

Human-readable and machine-verifiable structures will allow end users to sign more complex structures, apart from just string messages, and still be able to know exactly what they are signing (opposed to signing a bunch of arbitrary bytes).

Thus, in the future, the Cosmos signing message specification will be expected to expand upon its canonical JSON structure to include such functionality.

API

Application developers and designers should formalize a standard set of APIs that adhere to the following specification:

cosmosSignBytes

Params:

- `data` : the Cosmos signed message canonical JSON structure
- `address` : the Bech32 Cosmos account address to sign data with

Returns:

- `signature` : the Cosmos signature derived using signing algorithm `s`
-

Examples

Using the `secp256k1` as the DSA, `s` :

```
data = {  
    "@chain_id": "1",  
    "@type": "message",  
    "text": "I hereby claim I am ABC on Keybase!"  
}  
  
cosmosSignBytes(data, "cosmos1pvsch6cddahhrn5e8ekw0us50dpnugwnlfngt3")  
>  
"0x7fc4a495473045022100dec81a9820df0102381cdbf7e8b0f1e2cb64c58e0ecda1324543742e0388e41a
```

References

Addresses spec

- [Bech32](#)

Bech32 on Cosmos

The Cosmos network prefers to use the Bech32 address format wherever users must handle binary data. Bech32 encoding provides robust integrity checks on data and the human readable part (HRP) provides contextual hints that can assist UI developers with providing informative error messages.

In the Cosmos network, keys and addresses may refer to a number of different roles in the network like accounts, validators etc.

HRP table

HRP	Definition
cosmos	Cosmos Account Address
cosmosvalcons	Cosmos Validator Consensus Address
cosmosvaloper	Cosmos Validator Operator Address

Encoding

While all user facing interfaces to Cosmos software should exposed Bech32 interfaces, many internal interfaces encode binary value in hex or base64 encoded form.

To convert between other binary representation of addresses and keys, it is important to first apply the Amino encoding process before Bech32 encoding.

A complete implementation of the Amino serialization format is unnecessary in most cases. Simply prepending bytes from this [table](#) to the byte string payload before Bech32 encoding will sufficient for compatible representation.

Store

The store package defines the interfaces, types and abstractions for Cosmos SDK modules to read and write to Merkleized state within a Cosmos SDK application. The store package provides many primitives for developers to use in order to work with both state storage and state commitment. Below we describe the various abstractions.

Types

Store

The bulk of the store interfaces are defined [here](#), where the base primitive interface, for which other interfaces build off of, is the `Store` type. The `Store` interface defines the ability to tell the type of the implementing store and the ability to cache wrap via the `CacheWrapper` interface.

CacheWrapper & CacheWrap

One of the most important features a store has the ability to perform is the ability to cache wrap. Cache wrapping is essentially the underlying store wrapping itself within another store type that performs caching for both reads and writes with the ability to flush writes via `Write()`.

KVStore & CacheKVStore

One of the most important interfaces that both developers and modules interface with, which also provides the basis of most state storage and commitment operations, is the `KVStore`. The `KVStore` interface provides basic CRUD abilities and prefix-based iteration, including reverse iteration.

Typically, each module has its own dedicated `KVStore` instance, which it can get access to via the `sdk.Context` and the use of a pointer-based named key -- `KVStoreKey`. The `KVStoreKey` provides pseudo-OCAP. How exactly a `KVStoreKey` maps to a `KVStore` will be illustrated below through the `CommitMultiStore`.

Note, a `KVStore` cannot directly commit state. Instead, a `KVStore` can be wrapped by a `CacheKVStore` which extends a `KVStore` and provides the ability for the caller to execute `Write()` which commits state to the underlying state storage. Note, this doesn't actually flush writes to disk as writes are held in memory until `Commit()` is called on the `CommitMultiStore`.

`CommitMultiStore`

The `CommitMultiStore` interface exposes the top-level interface that is used to manage state commitment and storage by an SDK application and abstracts the concept of multiple `KVStore`s which are used by multiple modules. Specifically, it supports the following high-level primitives:

- Allows for a caller to retrieve a `KVStore` by providing a `KVStoreKey`.
- Exposes pruning mechanisms to remove state pinned against a specific height/version in the past.
- Allows for loading state storage at a particular height/version in the past to provide current head and historical queries.
- Provides the ability to rollback state to a previous height/version.
- Provides the ability to load state storage at a particular height/version while also performing store upgrades, which are used during live hard-fork application state migrations.
- Provides the ability to commit all current accumulated state to disk and performs Merkle commitment.

Implementation Details

While there are many interfaces that the `store` package provides, there is typically a core implementation for each main interface that modules and developers interact with that are defined in the Cosmos SDK.

`iavl.Store`

The `iavl.Store` provides the core implementation for state storage and commitment by implementing the following interfaces:

- `KVStore`
- `CommitStore`
- `CommitKVStore`
- `Queryable`
- `StoreWithInitialVersion`

It allows for all CRUD operations to be performed along with allowing current and historical state queries, prefix iteration, and state commitment along with Merkle proof operations. The `iavl.Store` also provides the ability to remove historical state from the state commitment layer.

An overview of the IAVL implementation can be found [here](#). It is important to note that the IAVL store provides both state commitment and logical storage operations, which comes with drawbacks as there are various performance impacts, some of which are very drastic, when it comes to the operations mentioned above.

When dealing with state management in modules and clients, the Cosmos SDK provides various layers of abstractions or "store wrapping", where the `iavl.Store` is the bottom most layer. When requesting a store to perform reads or writes in a module, the typical abstraction layer in order is defined as follows:

```
iavl.Store <- cachekv.Store <- gaskv.Store <- cachemulti.Store <- rootmulti.Store
```

Concurrent use of IAVL store

The tree under `iavl.Store` is not safe for concurrent use. It is the responsibility of the caller to ensure that concurrent access to the store is not performed.

The main issue with concurrent use is when data is written at the same time as it's being iterated over. Doing so will cause a irrecoverable fatal error because of concurrent reads and writes to an internal map.

Although it's not recommended, you can iterate through values while writing to it by disabling "FastNode" **without guarantees that the values being written will be returned during the iteration** (if you need this, you might want to reconsider the design of your application). This is done by setting `iavl-disable-fastnode` to `true` in the config TOML file.

cachekv.Store

The `cachekv.Store` store wraps an underlying `KVStore`, typically a `iavl.Store` and contains an in-memory cache for storing pending writes to underlying `KVStore`. `Set` and `Delete` calls are executed on the in-memory cache, whereas `Has` calls are proxied to the underlying `KVStore`.

One of the most important calls to a `cachekv.Store` is `Write()`, which ensures that key-value pairs are written to the underlying `KVStore` in a deterministic and ordered manner by sorting the keys first. The store keeps track of "dirty" keys and uses these to determine what keys to sort. In addition, it also keeps track of deleted keys and ensures these are also removed from the underlying `KVStore`.

The `cachekv.Store` also provides the ability to perform iteration and reverse iteration. Iteration is performed through the `cacheMergeIterator` type and uses both the dirty cache and underlying `KVStore` to iterate over key-value pairs.

Note, all calls to CRUD and iteration operations on a `cachekv.Store` are thread-safe.

gaskv.Store

The `gaskv.Store` store provides a simple implementation of a `KVStore`. Specifically, it just wraps an existing `KVStore`, such as a cache-wrapped `iavl.Store`, and incurs configurable gas costs for CRUD operations via `ConsumeGas()` calls defined on the `GasMeter` which exists in a `sdk.Context` and then proxies the underlying CRUD call to the underlying store. Note, the `GasMeter` is reset on each block.

cachemulti.Store & rootmulti.Store

The `rootmulti.Store` acts as an abstraction around a series of stores. Namely, it implements the `CommitMultiStore` and `Queryable` interfaces. Through the `rootmulti.Store`, an SDK module can request access to a `KVStore` to perform state CRUD operations and queries by holding access to a unique `KVStoreKey`.

The `rootmulti.Store` ensures these queries and state operations are performed through cached-wrapped instances of `cachekv.Store` which is described above. The `rootmulti.Store` implementation

is also responsible for committing all accumulated state from each `KVStore` to disk and returning an application state Merkle root.

Queries can be performed to return state data along with associated state commitment proofs for both previous heights/versions and the current state root. Queries are routed based on store name, i.e. a module, along with other parameters which are defined in `abci.RequestQuery`.

The `rootmulti.Store` also provides primitives for pruning data at a given height/version from state storage. When a height is committed, the `rootmulti.Store` will determine if other previous heights should be considered for removal based on the operator's pruning settings defined by `PruningOptions`, which defines how many recent versions to keep on disk and the interval at which to remove "staged" pruned heights from disk. During each interval, the staged heights are removed from each `KVStore`. Note, it is up to the underlying `KVStore` implementation to determine how pruning is actually performed. The `PruningOptions` are defined as follows:

```
type PruningOptions struct {
    // KeepRecent defines how many recent heights to keep on disk.
    KeepRecent uint64

    // Interval defines when the pruned heights are removed from disk.
    Interval uint64

    // Strategy defines the kind of pruning strategy. See below for more information
    // on each.
    Strategy PruningStrategy
}
```

The Cosmos SDK defines a preset number of pruning "strategies": `default`, `everything`, `nothing`, and `custom`.

It is important to note that the `rootmulti.Store` considers each `KVStore` as a separate logical store. In other words, they do not share a Merkle tree or comparable data structure. This means that when state is committed via `rootmulti.Store`, each store is committed in sequence and thus is not atomic.

In terms of store construction and wiring, each Cosmos SDK application contains a `BaseApp` instance which internally has a reference to a `CommitMultiStore` that is implemented by a `rootmulti.Store`. The application then registers one or more `KVStoreKey` that pertain to a unique module and thus a `KVStore`. Through the use of an `sdk.Context` and a `KVStoreKey`, each module can get direct access to its respective `KVStore` instance.

Example:

```
func NewApp(...) Application {
    // ...

    bApp := baseapp.NewBaseApp(appName, logger, db, txConfig.TxDecoder(),
    baseAppOptions...)
    bApp.SetCommitMultiStoreTracer(traceStore)
    bApp.SetVersion(version.Version)
    bApp.SetInterfaceRegistry(interfaceRegistry)
```

```

// ...

keys := sdk.NewKVStoreKeys(...)
transientKeys := sdk.NewTransientStoreKeys(...)
memKeys := sdk.NewMemoryStoreKeys(...)

// ...

// initialize stores
app.MountKVStores(keys)
app.MountTransientStores(transientKeys)
app.MountMemoryStores(memKeys)

// ...
}

```

The `rootmulti.Store` itself can be cache-wrapped which returns an instance of a `cachemulti.Store`. For each block, `BaseApp` ensures that the proper abstractions are created on the `CommitMultiStore`, i.e. ensuring that the `rootmulti.Store` is cached-wrapped and uses the resulting `cachemulti.Store` to be set on the `sdk.Context` which is then used for block and transaction execution. As a result, all state mutations due to block and transaction execution are actually held ephemerally until `Commit()` is called by the ABCI client. This concept is further expanded upon when the AnteHandler is executed per transaction to ensure state is not committed for transactions that failed CheckTx.

Inter-block Cache

- [Inter-block Cache](#)
 - [Synopsis](#)
 - [Overview and basic concepts](#)
 - [Motivation](#)
 - [Definitions](#)
 - [System model and properties](#)
 - [Assumptions](#)
 - [Properties](#)
 - [Thread safety](#)
 - [Crash recovery](#)
 - [Iteration](#)
 - [Technical specification](#)
 - [General design](#)
 - [API](#)
 - [CommitKVCacheManager](#)
 - [CommitKVStoreCache](#)
 - [Implementation details](#)
 - [History](#)
 - [Copyright](#)

Synopsis

The inter-block cache is an in-memory cache storing (in-most-cases) immutable state that modules need to read in between blocks. When enabled, all sub-stores of a multi store, e.g., `rootmulti`, are wrapped.

Overview and basic concepts

Motivation

The goal of the inter-block cache is to allow SDK modules to have fast access to data that it is typically queried during the execution of every block. This is data that do not change often, e.g. module parameters. The inter-block cache wraps each `CommitKVStore` of a multi store such as `rootmulti` with a fixed size, write-through cache. Caches are not cleared after a block is committed, as opposed to other caching layers such as `cachekv`.

Definitions

- `Store key` uniquely identifies a store.
- `KVCache` is a `CommitKVStore` wrapped with a cache.
- `Cache manager` is a key component of the inter-block cache responsible for maintaining a map from `store keys` to `KVCaches`.

System model and properties

Assumptions

This specification assumes that there exists a cache implementation accessible to the inter-block cache feature.

The implementation uses adaptive replacement cache (ARC), an enhancement over the standard last-recently-used (LRU) cache in that tracks both frequency and recency of use.

The inter-block cache requires that the cache implementation to provide methods to create a cache, add a key/value pair, remove a key/value pair and retrieve the value associated to a key. In this specification, we assume that a `Cache` feature offers this functionality through the following methods:

- `NewCache(size int)` creates a new cache with `size` capacity and returns it.
- `Get(key string)` attempts to retrieve a key/value pair from `Cache`. It returns `(value []byte, success bool)`. If `Cache` contains the key, it `value` contains the associated value and `success=true`. Otherwise, `success=false` and `value` should be ignored.
- `Add(key string, value []byte)` inserts a key/value pair into the `Cache`.
- `Remove(key string)` removes the key/value pair identified by `key` from `Cache`.

The specification also assumes that `CommitKVStore` offers the following API:

- `Get(key string)` attempts to retrieve a key/value pair from `CommitKVStore`.
- `Set(key, string, value []byte)` inserts a key/value pair into the `CommitKVStore`.
- `Delete(key string)` removes the key/value pair identified by `key` from `CommitKVStore`.

Ideally, both `Cache` and `CommitKVStore` should be specified in a different document and referenced here.

Properties

Thread safety

Accessing the `cache manager` or a `KVCache` is not thread-safe: no method is guarded with a lock. Note that this is true even if the cache implementation is thread-safe.

For instance, assume that two `Set` operations are executed concurrently on the same key, each writing a different value. After both are executed, the cache and the underlying store may be inconsistent, each storing a different value under the same key.

Crash recovery

The inter-block cache transparently delegates `Commit()` to its aggregate `CommitKVStore`. If the aggregate `CommitKVStore` supports atomic writes and use them to guarantee that the store is always in a consistent state in disk, the inter-block cache can be transparently moved to a consistent state when a failure occurs.

Note that this is the case for `IAVLStore`, the preferred `CommitKVStore`. On commit, it calls `SaveVersion()` on the underlying `MutableTree`. `SaveVersion` writes to disk are atomic via batching. This means that only consistent versions of the store (the tree) are written to the disk. Thus, in case of a failure during a `SaveVersion` call, on recovery from disk, the version of the store will be consistent.

Iteration

Iteration over each wrapped store is supported via the embedded `CommitKVStore` interface.

Technical specification

General design

The inter-block cache feature is composed by two components: `CommitKVCacheManager` and `CommitKVCache`.

`CommitKVCacheManager` implements the cache manager. It maintains a mapping from a store key to a `KVStore`.

```
type CommitKVStoreCacheManager interface{
    cacheSize uint
    caches map[string]CommitKVStore
}
```

`CommitKVStoreCache` implements a `KVStore`: a write-through cache that wraps a `CommitKVStore`. This means that deletes and writes always happen to both the cache and the underlying `CommitKVStore`. Reads on the other hand first hit the internal cache. During a cache miss, the read is delegated to the underlying `CommitKVStore` and cached.

```
type CommitKVStoreCache interface{
    store CommitKVStore
    cache Cache
}
```

To enable inter-block cache on `rootmulti`, one needs to instantiate a `CommitKVCacheManager` and set it by calling `SetInterBlockCache()` before calling one of `LoadLatestVersion()`,

```
LoadLatestVersionAndUpgrade(...), LoadVersionAndUpgrade(...) and LoadVersion(version).
```

API

CommitKVCacheManager

The method `NewCommitKVStoreCacheManager` creates a new cache manager and returns it.

Name	Type	Description
size	integer	Determines the capacity of each of the KVCache maintained by the manager

```
func NewCommitKVStoreCacheManager(size uint) CommitKVStoreCacheManager {
    manager = CommitKVStoreCacheManager{size, make(map[string]CommitKVStore)}
    return manager
}
```

`GetStoreCache` returns a cache from the `CommitStoreCacheManager` for a given store key. If no cache exists for the store key, then one is created and set.

Name	Type	Description
manager	CommitKVStoreCacheManager	The cache manager
storeKey	string	The store key of the store being retrieved
store	CommitKVStore	The store that it is cached in case the manager does not have any in its map of caches

```
func GetStoreCache(
    manager CommitKVStoreCacheManager,
    storeKey string,
    store CommitKVStore) CommitKVStore {

    if manager.caches.has(storeKey) {
        return manager.caches.get(storeKey)
    } else {
        cache = CommitKVStoreCacheManager{store, manager.cacheSize}
        manager.set(storeKey, cache)
        return cache
    }
}
```

`Unwrap` returns the underlying `CommitKVStore` for a given store key.

Name	Type	Description
manager	CommitKVStoreCacheManager	The cache manager
storeKey	string	The store key of the store being unwrapped

```

func Unwrap(
    manager CommitKVStoreCacheManager,
    storeKey string) CommitKVStore {

    if manager.caches.has(storeKey) {
        cache = manager.caches.get(storeKey)
        return cache.store
    } else {
        return nil
    }
}

```

`Reset` resets the manager's map of caches.

Name	Type	Description
manager	CommitKVStoreCacheManager	The cache manager

```

function Reset(manager CommitKVStoreCacheManager) {

    for (let storeKey of manager.caches.keys()) {
        manager.caches.delete(storeKey)
    }
}

```

CommitKVStoreCache

`NewCommitKVStoreCache` creates a new `CommitKVStoreCache` and returns it.

Name	Type	Description
store	CommitKVStore	The store to be cached
size	string	Determines the capacity of the cache being created

```

func NewCommitKVStoreCache(
    store CommitKVStore,
    size uint) CommitKVStoreCache {
    KVCache = CommitKVStoreCache{store, NewCache(size)}
    return KVCache
}

```

`Get` retrieves a value by key. It first looks in the cache. If the key is not in the cache, the query is delegated to the underlying `CommitKVStore`. In the latter case, the key/value pair is cached. The method returns the value.

Name	Type	Description
KVCache	CommitKVStoreCache	The <code>CommitKVStoreCache</code> from which the key/value pair is retrieved

key	string	Key of the key/value pair being retrieved
-----	--------	---

```
func Get(
    KVCache CommitKVStoreCache,
    key string) []byte {
    valueCache, success := KVCache.cache.Get(key)
    if success {
        // cache hit
        return valueCache
    } else {
        // cache miss
        valueStore = KVCache.store.Get(key)
        KVCache.cache.Add(key, valueStore)
        return valueStore
    }
}
```

`Set` inserts a key/value pair into both the write-through cache and the underlying `CommitKVStore`.

Name	Type	Description
KVCache	CommitKVStoreCache	The <code>CommitKVStoreCache</code> to which the key/value pair is inserted
key	string	Key of the key/value pair being inserted
value	[]byte	Value of the key/value pair being inserted

```
func Set(
    KVCache CommitKVStoreCache,
    key string,
    value []byte) {

    KVCache.cache.Add(key, value)
    KVCache.store.Set(key, value)
}
```

`Delete` removes a key/value pair from both the write-through cache and the underlying `CommitKVStore`.

Name	Type	Description
KVCache	CommitKVStoreCache	The <code>CommitKVStoreCache</code> from which the key/value pair is deleted
key	string	Key of the key/value pair being deleted

```
func Delete(
    KVCache CommitKVStoreCache,
    key string) {

    KVCache.cache.Remove(key)
```

```
KVCache.store.Delete(key)  
}
```

`CacheWrap` wraps a `CommitKVStoreCache` with another caching layer (`CacheKV`).

It is unclear whether there is a use case for `CacheWrap`.

Name	Type	Description
<code>KVCache</code>	<code>CommitKVStoreCache</code>	The <code>CommitKVStoreCache</code> being wrapped

```
func CacheWrap(  
    KVCache CommitKVStoreCache) {  
  
    return CacheKV.NewStore(KVCache)  
}
```

Implementation details

The inter-block cache implementation uses a fixed-sized adaptive replacement cache (ARC) as cache. [The ARC implementation](#) is thread-safe. ARC is an enhancement over the standard LRU cache in that tracks both frequency and recency of use. This avoids a burst in access to new entries from evicting the frequently used older entries. It adds some additional tracking overhead to a standard LRU cache, computationally it is roughly $2\times$ the cost, and the extra memory overhead is linear with the size of the cache. The default cache size is 1000.

History

Dec 20, 2022 - Initial draft finished and submitted as a PR

Copyright

All content herein is licensed under [Apache 2.0](#).